Universidad
Carlos III de Madrid
www.uc3m.es

TRABAJO FIN DE GRADO

Estimación de zonas cruzables utilizando el sensor Kinect.

Autor: Adrián García López

Titulación: Grado en Electrónica Industrial y Automática

Profesor: Fernando M. Monar

Fecha: Septiembre 2012

Resumen:

La búsqueda de robots o sistemas automáticos que puedan moverse por entornos por los que podemos pasar los humanos es muy importante. Para disponer de una movilidad íntegra que les permitan desempeñar trabajos e imitar las actividades de los seres humanos estos sistemas deberán disponer un software y un hardware que les permita desarrollar dichas acciones.

Este trabajo está centrado en el diseño de un software que, mediante la interpretación de imágenes tomadas con la cámara Kinect de Microsoft, sea capaz de estimar los planos por los que un robot o sistema equipado con este dispositivo podría transitar sin el riesgo de colisionar con ningún objeto.

Mediante la toma de nubes de puntos con el sensor Kinect, evaluaremos los puntos que estén dentro del plano formado por el suelo. Posteriormente obtendremos los puntos correspondientes a obstáculos posibles, los cuales proyectaremos y eliminaremos de la superficie del suelo obtenida anteriormente para dar con un resultado válido.

Gracias a este dispositivo y al programa diseñado podremos encontrar las zonas cruzables de estancias con objetos, pudiendo incluso determinarlas en condiciones adversas como puede ser la poca/nula visibilidad.

ABSTRACT:

Many mobile robots need to move freely in environments designed for human beings. These systems must have a software and hardware that allow them to execute different tasks in these environments.

This work is focused on designing a program that, through the interpretation of images taken with the Microsoft Kinect camera, estimates the surfaces where a robot equipped with this device can travel without any risk of collision.

First, the points that belong to the floor are extracted from the initial point cloud. After that, the possible obstacles are estimated and removed and the traversable zones are computed. The designed program allows us to find free areas without objects and can even determine this in adverse conditions such as the low or zero visibility.

Índice:

Índice de figuras:

1. Introducción

La idea de utilizar los robots como sustitutos de los humanos en diferentes tareas está consolidada desde los inicios de la robótica. Dentro de estas funciones podemos encontrar el uso para tareas triviales y sencillas tales como la limpieza del suelo de una estancia hasta para tareas 3D (dull, dirty and dangerous) como podría ser la inspección de suelo lunar. En la Figura 1 podemos ver un robot desactivador de bombas, una de las muchas tareas peligrosas que se puede realizar con robots.

Figura 1: Robot Packbot desactivador de bombas.

Dichas tareas requieren robots que puedan reconocer el entorno por el que se mueven esquivando objetos y obstáculos que se presenten en su trayectoria a través de un aparato que les pueda proporcionar visión.

Para llevar a cabo una correcta percepción del entorno se deberá tener un elemento que nos proporcione la información del exterior correctamente con un corto o nulo margen de error. Además deberá disponer de un software que permita procesar toda la información obtenida por nuestro elemento de visión para que posteriormente pueda actuar de una manera coherente en función del objetivo asignado.

La visión artificial no es algo trivial ya que debemos tener la capacidad de percibir nuestro entorno en un mundo dinámico que sufre constantes variaciones y movimientos, cambios de iluminación y perdida de información por puntos muertos [1, 2]. Existen múltiples formas de llevar a cabo un reconocimiento del entorno. Actualmente los sistemas modernos se centran principalmente en la proyección de haces laser o una luz infrarroja midiendo el tiempo de vuelo, es decir, lo que tarda el haz en llegar al objeto o entorno y volver. También se puede medir la diferencia de fase de la onda enviada y la recibida para obtener la distancia.

Dentro del abanico de aplicaciones para las que se pueden utilizar este tipo de sensores podemos encontrar el reconocimiento facial como el mostrado en la Figura 2, reconocimiento de objetos, formas, colores además de análisis de distancias y tamaños entre otros. Estas habilidades incorporadas a un humanoide les proporcionarían la posibilidad de reconocer objetos, saber si puede manipularlos, analizar por donde debería de cruzar una estancia para no chocar con ningún obstáculo, etc.

Figura 2: Ejemplo de reconocimiento facial en una nube de puntos.

En el presente proyecto se presentará la realización de un software que permite detectar las zonas transitables libres de obstáculos para un robot tipo estimado con la finalidad de poder diferenciar cuales son las zonas en las que dicho robot se podría situar.

Todo esto se realizara mediante una cámara KINECT (Sección 3.3) de Microsoft dentro del sistema operativo Ubuntu 11.10 además del uso de diferentes bibliotecas PCL y OpenNI que describiremos con más detenimiento posteriormente en el Capítulo 4.

El siguiente documento estará distribuido en distintos capítulos. Inicialmente, en el Capítulo 2 tenemos el estado del arte donde expondremos distintos dispositivos que obtienen mediante sensores el entorno que les rodea. En el Capítulo 3 veremos distintos dispositivos de visión haciendo hincapié en el sensor Kinect. Veremos las bibliotecas y el lenguaje utilizado en el Capítulo 4. Se pasará a describir el funcionamiento del programa paso a paso a lo largo del Capítulo 5 y posteriormente se presentará un listado de pruebas en el Capítulo 6. Por último, se llevarán a cabo las conclusiones y el análisis de posibles mejoras en el Capítulo 7.

1.1. Objetivo del proyecto

El objetivo principal del proyecto es el diseño de un software compuesto de diferentes algoritmos que permitan:

- Creación de nubes de puntos del entorno mediante capturas en 3D utilizando una cámara Kinect.
- Extracción de las zonas de interés de esas nubes de puntos relativas al proyecto para su posterior estudio.
- Segmentación del plano correspondiente al suelo libre de objetos por el que transitará un robot predefinido.

Todo ello lo haremos de manera modular para dar mayor versatilidad al programa y una gran robustez.
Utilizaremos como lenguaje de programación C++ y nos apoyaremos principalmente en las librerías de PCL.

2. Estado del Arte

Este proyecto gira entorno a la percepción de entornos y de la estimación de zonas cruzables por lo que en el siguiente apartado expondremos de manera general algunos ejemplos de aplicación de dichos dispositivos pasando después por los sensores existentes y por último presentando el sensor utilizado.

Existen muchos grupos de investigación desde finales del siglo XX, dedicados al desarrollo de robots capaces de transitar y desenvolverse con facilidad en distintos entornos, tanto exteriores como interiores (proyecto UGV [3], proyecto EDEN [4]. Actualmente se sigue trabajando en perfeccionar las capacidades y la autonomía de estos robots que dependerán de las aplicaciones para las que se emplearán. Estas aplicaciones pueden ser:

- Navegación autónoma terrestre:
  - por carreteras, caminos o entornos parcialmente delimitados y/o conocidos.
  - por zonas no conocidas ni delimitadas.
- Navegación en terrenos ajenos o distintos al terrestre (exploración planetaria o lunar).
- Navegación en lugares peligrosos o poco accesibles (minas, centrales nucleares).

A continuación presentaremos una serie de ejemplos sobre el desarrollo de robots capaces de transitar de una manera autónoma e independiente gracias a una serie de sensores con los que captan el entorno que les rodea.

2.1. STANLEY

Es el claro ejemplo de la búsqueda de vehículos autónomos no tripulados resultado de la prueba Grand Challenge 2005 promovida por DARPA (organización de investigación del Departamento de Defensa de los Estados Unidos) [5].

Utilizando el Volkswagen Touareg de la Figura 3 como base, se le incorporaron 6 Placas Pentium M en un bastidor junto a módulos de software personalizados para planificación y optimización, control, navegación, procesamiento de la información obtenida por los sensores, etc.

Figura 3: Proyecto Stanley, ganador del DARPA Grand Challenge 2005.

El sistema de navegación estaba formado por:

- Vista:
  - Cinco telémetros láser (25m).
  - Cámara de video monocular a color (150m).
  - Radar de largo alcance (200m).
- Posición:
  - Sensor GPS con una resolución de 20 cm.
  - Sensores de velocidad en las ruedas.
- Balance:
  - Unidad de medición inercial que proporciona 6 grados de libertad.
  - Brújula GPS que genera dos grados de libertad.

2.2 CURIOSITY

Robot rover (Figura 4) enviado a la superficie de Marte que actualmente se encuentra explorando la superficie marciana en busca de indicios de vida en el planeta vecino [6]. Dispone de 6 ruedas independientemente alimentadas con una suspensión que le permite transitar sin problemas por espacios rugosos o con pequeños cráteres. Además dispone de sistemas de procesamiento y memoria destinados a controlar tanto el robot como el entorno capaces de soportar las radiaciones extremas a las que se enfrentara en el espacio. La parte más relevante para nuestro trabajo es la capacidad sensorial que dispone el robot compuesta por:

Figura 4: Robot Curiosity enviado a la superficie de Marte.

- Cuatro pares de cámaras para evitar peligro (Hazcams):

Montadas por todo el vehículo estas cámaras permiten la captura del entorno para su posterior procesamiento buscando evitar colisiones. Cada cámara dispone de un haz de visión de 120 grados y 3 metros de profundidad. Estas cámaras están fijas al cuerpo robótico.

- Dos pares de cámaras de navegación (Navcams):

Montadas en el mástil que corresponderían al "cuello y la cabeza" del rover, estas cámaras en blanco y negro usan la luz visible para reunir una panorámica en 3D. Cada una de ellas apoya la planificación de la navegación en el suelo, aunque será apoyado por los científicos e ingenieros que utilizando las Hazcams proporcionarán una visión complementaria del terreno.

- Cuatro cámaras científicas (un par de Mastcam, Chemcam y Mahli):

Estas cuatro cámaras estarán dedicadas al estudio del suelo lunar, tanto minerales como microestructuras de las rocas.

- Cámara de Descenso (Mardi)

Dedicado íntegramente a la detección del movimiento de la nave para su posterior ajuste (en caso de que fuese necesario) con los retrocohetes.

2.3. MDARS

El robot MDARS [7] (Mobile Detection Assessment Response Systems) como el que podemos ver en la Figura 5, fue creado por iniciativa del Ejercito y la Marina de los Estados Unidos para proporcionar un sistema destinado al control de centros de almacenamiento del departamento de defensa además de proporcionar múltiples plataformas móviles que llevasen a cabo patrullas aleatorias dentro de putos clave para el gobierno norteamericano.

Figura 5: Robot MDARS patrullando los alrededores de una central nuclear.

Robot de base compuesta por 4 ruedas motrices movidas mediante un motor diesel (con 16 horas de autonomía) que cuenta con sensores laser, de ultrasonidos, sonar, radar de ondas milimétricas además de visión estéreo para evitar colisiones. Desarrolla su navegación autónoma bajo trazados establecidos gracias a su GPS. Aunque disponga de multitud de sensores capaces de proporcionarle un mapeado de su entorno no es un robot capaz de estimar trayectorias de manera autónoma, únicamente es capaz de evitar obstáculos. Actualmente son utilizados para proteger distintas áreas o edificios tales como centrales nucleares o centros penitenciarios.

2.4. MANFRED-2

El robot MANFRED-2 [8], totalmente desarrollado por el Laboratorio de Robótica de la Universidad Carlos III de Madrid, es un manipulador móvil construido con la finalidad de poder navegar de forma autónoma.

Su diseño ha sido inspirado en los rovers planetarios y los satélites de comunicaciones, todo de una manera muy modular centrada en unidades independientes interconectadas mediante el uso de interfaces eléctricas y mecánicas. En la Figura 6 podemos ver una foto de este robot.

Figura 6: Robot MANFRED-2

Dentro de los sensores que le proporcionan la información necesaria para poder determinar los espacios transitables de manera autónoma se encuentran:

- Laser Hokuyo UTM-30LX que permite obtener un mapeado del entorno.
- Laser SICK PLS empleado también para mapear el entorno.
- Cámara SONY EVI- D100 empleada para reconocer objetos y estimar sus posiciones relativas frente al robot.
- Cámara SONY B/N XC-ES50CE. Utilizada para tareas de manipulación, aunque es un recurso auxiliar cuando la cámara SONY EVI no dispone de visión por obstrucción.
- Cámara Kinect: Gracias a esta cámara se pueden obtener información de la forma del objeto además de la distancia que le separa.

3. Dispositivos de visión

En este capítulo se engloban diferentes dispositivos de visión que podemos encontrar desde el ojo humano pasado por cámaras, sensores laser, etc. Haremos hincapié en el sensor Kinect utilizado en este proyecto y terminaremos con la cámara Asus que intenta imitar a nuestro objeto de trabajo.

3.1 El Ojo Humano

Todos los anteriores ejemplos de dispositivos de navegación autónoma disponían de multitud de sensores que intentaban igualar la capacidad de visión del ser humano. Esta capacidad nos es proporcionada gracias al ojo (Figura 7), nuestra base del sentido de la vista, que nos permite desenvolvernos con suma facilidad en nuestra vida diaria. A diferencia de los grandes dispositivos que componían los sistemas anteriores, el ojo humano es claramente inferior ya que tiene forma aproximadamente esférica de unos 2, 5 cm de diámetro que funciona de manera muy similar al de la mayoría de los vertebrados.

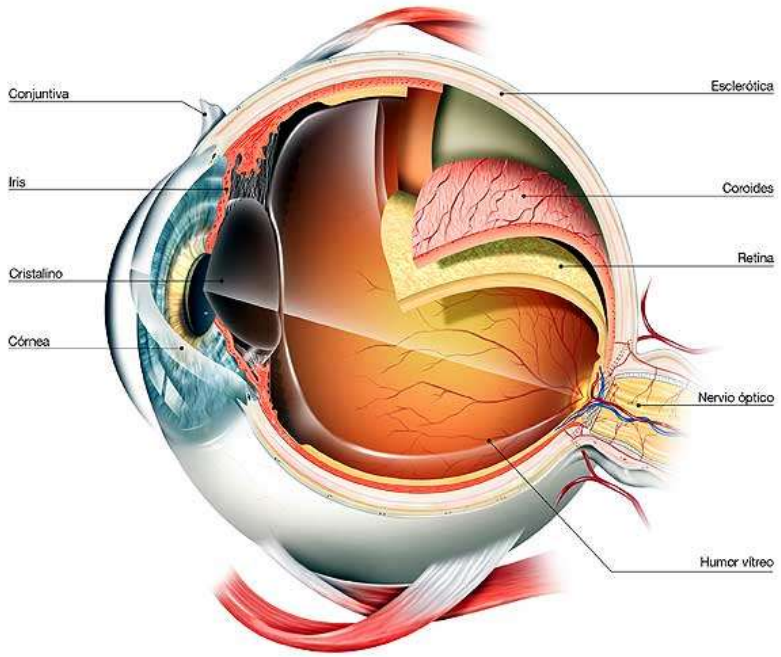


Figura 7: El ojo humano.

Posee un lente llamado cristalino que es ajustable según la distancia, un diafragma que se llama pupila cuyo diámetro está regulado por el iris y un tejido sensible a la luz que es la retina. La luz penetra a través de la pupila, atraviesa el cristalino y se proyecta sobre la retina, donde se transforma gracias a unas células llamadas fotorreceptoras en impulsos nerviosos que son trasladados a través del nervio óptico al cerebro.

3.2. Dispositivos de visión actuales

Actualmente la visión robótica se centra en sensores láser e infrarrojos que captan la profundidad de los objetos pudiendo posteriormente crear un mallado del entorno. Tienen un papel muy importante las cámaras [9].

El campo de la visión en la robótica siempre ha intentado crear algún dispositivo que se asimile a este órgano humano por completo en cuanto a capacidad y movimientos para poder proporcionar esa habilidad a los robots existentes. Es muy llamativa la cámara Superfast mostrada en la Figura 8, dispositivo creado por investigadores alemanes que simula los movimientos sacadicos del ojo humano pudiendo además alcanzar cualquier punto de visionado.

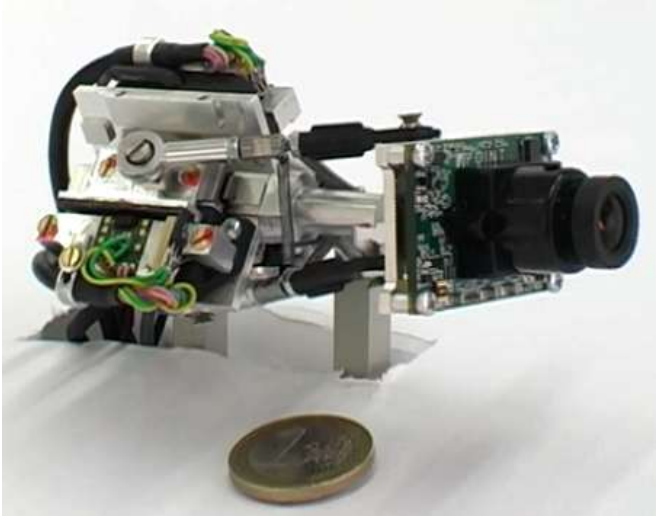


Figura 8: Cámara Superfast.

Mediante un sistema paralelo que utiliza actuadores piezoeléctricos que alcanzan velocidades muy altas cubre el rango angular de visión del ojo humano. Los componentes de esta cámara podemos apreciarlos en la Figura 9 .El trabajo ha sido dirigido por el profesor Heinz Ulbrich de la Universidad de Múnich.

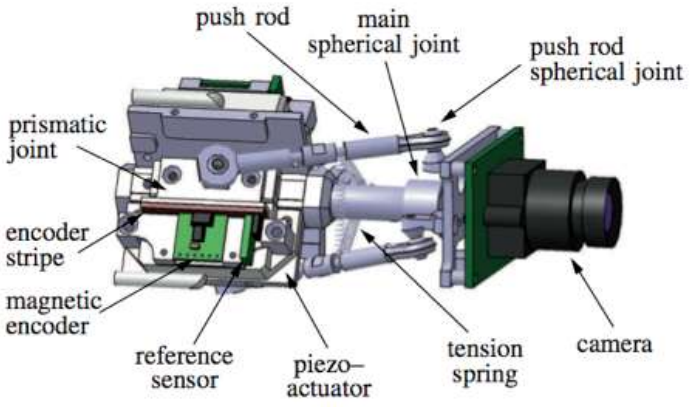


Figura 9: Partes de la cámara Superfast.

En nuestro proyecto, más que asimilar el ojo humano en cuanto a movimientos, buscamos poder captar de manera fehaciente el entorno, captando la medida de profundidades de lo que nos rodea permitiéndonos crear una imagen virtual.

En los últimos años los sensores más utilizados han sido las cámaras y los sensores laser (Figura 11). Estos últimos nos permiten realizar un escaneado (Figura 10) que nos obtenga una nube de puntos a partir de muestras de la superficie del entorno. Posteriormente esos puntos pueden ser evaluados para estudiar la forma del objeto, información del entorno, etc.

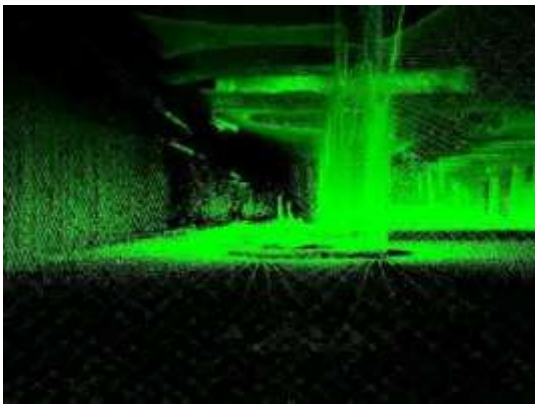


Figura 10: Ejemplo de mapeado con un dispositivo laser.

Estos láseres en 3D a pesar de disponer de un campo de visión limitado al igual que las cámaras, no reúnen información acerca del su color de los objetos, únicamente captan la geometría. El modelo obtenido por un escáner 3D describe la posición en el espacio tridimensional de cada punto analizado.

Usualmente los láseres disponen de un campo de visión cónico aunque existen dispositivos que sólo lanzan una línea por lo que es común equiparlos con una unidad motorizada que les proporcione un grado de visión adicional. Existen numerosos tipos en función de si requieren contacto o no, y si son activos o pasivos. Gracias a este amplio abanico de posibilidades se puede disponer de un dispositivo de este tipo acorde con los requerimientos de nuestro trabajo

Figura 11: Escáner Laser Hokuyo UTM 30-LX.

3.3. Microsoft Kinect

Uno de los problemas de los sensores láser es su elevado precio. En el mercado encontramos un sensor muy versátil y de reducido precio (en torno a 150$) como es Kinect, sensor desarrollado por Microsoft para la videoconsola Xbox 360 mostrado en la Figura 12.

Dicho dispositivo se diseñó con el fin de permitir a los usuarios de estas consolas controlar e interactuar con estas desechando todo mando o medio de contacto físico, utilizando una interfaz que permita controlar todas las opciones del videojuego mediante los gestos de nuestro cuerpo y comandos de voz.

Microsoft anunció Kinect por primera vez el 1 de junio de 2009 con el "Proyect Natal", que vaticinaba una interacción sin precedentes con una videoconsola olvidando así los mandos por completo. Finalmente fue lanzado al mercado el 4 de Noviembre de 2010 en Norteamérica.

Figura 12: Cámara Kinect.

Kinect se puede dividir en una base circular con una rótula que dispone de un mecanismo de inclinación motorizado y por otro lado la barra horizontal que contiene una cámara RGB, un sensor de profundidad, un micrófono y un procesador personalizado (PrimeSense PS1080-A2) que ejecuta el software patentado de Microsoft. Dicha barra tiene unas medidas de 28 cm de largo, 7,5 de ancho y 7 de alto que son aproximadas ya que no es un prisma exacto debido a que la parte delantera de la cámara es un poco mas alta que la parte posterior.

El micrófono de matrices del sensor de Kinect permite a la Xbox 360 llevar a cabo la localización de la fuente acústica y supresión del ruido ambiente gracias a su proyección cónica.

La cámara detecta colores clasificándolos en un código RGB en función del porcentaje que compongan estos. Además dispone de la capacidad de capturar 30fps con una resolución 640x480 VGA.

El sensor de profundidad es un proyector de haz infrarrojo y un sensor CMOS monocromo que permite a Kinect ver la habitación en 3D ante cualquier condición de luz ambiental. El rango de detección de la profundidad es ajustable ya que se puede calibrar automáticamente el sensor ante la variabilidad de las estancias de juego. Esto no quiere decir que el sensor tenga una capacidad alta, ya que para condiciones óptimas de captura se han de cumplir unas condiciones de campo de visión optimas:

- Entre 1,2m-3,5m de distancia visión óptima
- Campo de visión horizontal de 57º
- Campo de visión vertical de 43º
- Rango de inclinación física: ± 27 grados divididos en 10 posiciones

La distancia ideal de trabajo recomendada por Microsoft se encuentra a 2,5metros y a distancias superiores a 3,5 metros comenzaremos a tener unas capturas muy deficientes las cuales serán muy pobres en información. En la Figura 13 podemos ver un grafico que representa el error cometido por nuestra cámara en función de la distancia que se desee capturar.

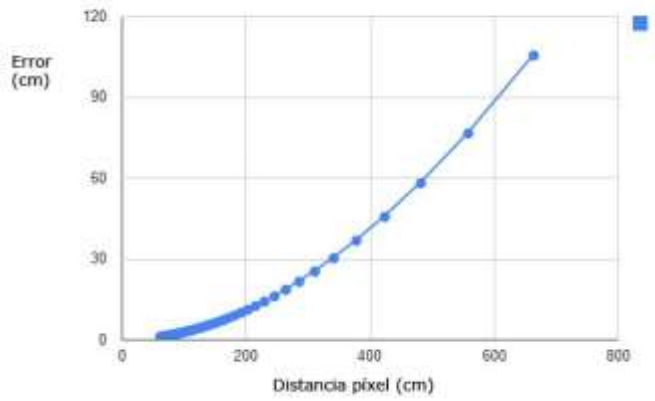


Figura 13: Gráfico del error cometido por la cámara Kinect.

Como podemos observar el error cometido a una distancia superior a 2 metros comienza a ser un error muy grande que nos puede originar bastantes problemas de cálculo, más teniendo en cuenta el problema al que nos enfrentamos que requiere un alto grado de precisión.
Pasando a la parte de hardware, Kinect se basa en un diseño de referencia y la tecnología 3D-calor fabricados por la compañía israelí de desarrollo PrimeSense Ltd.

Por último la alimentación de este dispositivo se lleva a cabo mediante un cable USB en el caso del modelo Xbox 360s (ya que está dotado con una toma de corriente diseñada para alimentar correctamente a Kinect) o en su defecto deberá ser conectado a una toma de corriente requiriendo 12V y 1,08A en CC. La transmisión de datos se llevará a cabo mediante un cable USB.

En la Figura 14 podemos explicar gráficamente como funciona Kinect:

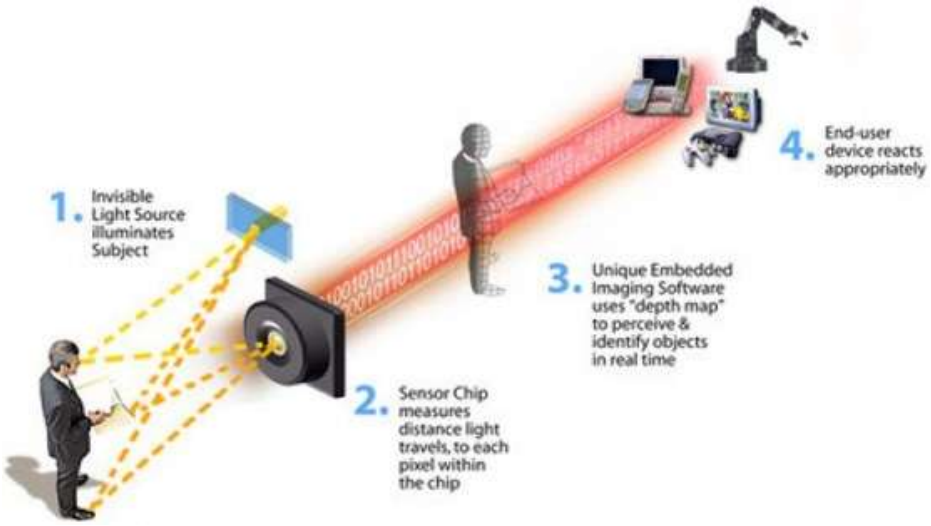


Figura 14: Ejemplo de funcionamiento de la cámara Kinect.

Como podemos observar la cámara emite un haz infrarrojo (Figura 15) invisible para el ojo humano con la que podremos medir la profundidad del ambiente. Con el receptor CMOS detecta esta luz para poder medir distancias y generar un mapeado del entorno. El funcionamiento es similar al de un sensor sonar salvo que utilizaremos en este caso luz infrarroja en lugar de sonido. Todo esto combinado con la captación de colores permite una diferenciación muy fehaciente del entorno.

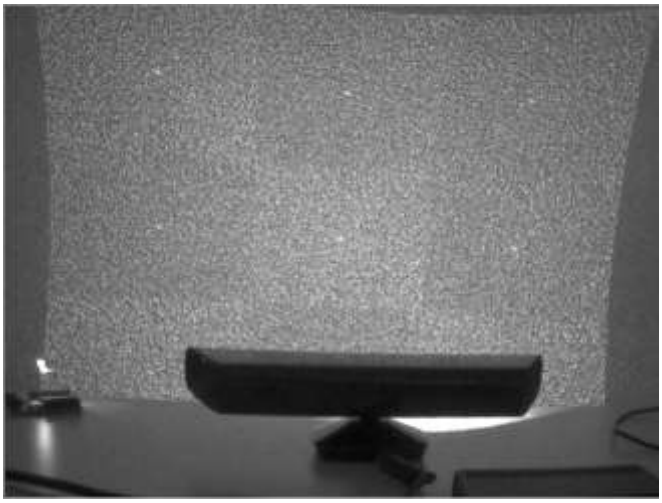


Figura 15: Proyección infrarroja con la cámara Kinect.

3.4. Asus Xtion Pro

Ante la potencialidad que presenta el dispositivo Kinect, el mercado ha visto la entrada de nuevos sensores similares de distintas marcas como es el caso de Asus, quien presenta su sensor de movimiento Xtion Pro (Figura 16).

Figura 16: Asus Xtion Pro, principal competidor del sensor Kinect.

Como se puede apreciar en la Figura 17, es un reflejo de Kinect en cuanto a forma se refiere, pero además si miramos detenidamente los componentes de dicho dispositivo veremos hasta qué punto es similar a Kinect.

Figura 17: Componentes de Asus Xtion Pro.

Dispone de una cámara VGA, sensor infrarrojo y de profundidad además de dos micrófonos frontales. Es un sistema que, a diferencia de Kinect, está pensado principalmente para el desarrollo de juegos y aplicaciones que requieran de dichos dispositivos. Además incorpora un software creado en parte por los también creadores del software de Kinect, PrimeSense, que incorpora juegos y aplicaciones para interactuar con redes sociales o juegos que podemos adquirir en la tienda Asus.

A diferencia de la Kinect, este dispositivo es alimentado únicamente mediante USB (la Kinect en función del puerto puede requerir alimentación añadida por un enchufe). Hemos de resaltar que este dispositivo es más caro que la Kinect.

4. Herramientas de programación

En este capítulo veremos las diferentes librerías utilizadas para la realización de nuestro programa pasando también por el lenguaje de programación utilizado. Concluiremos con las funciones y clases utilizadas en la realización del programa que se detallara posteriormente en el Capítulo 5.

4.1. OpenNI

Para la realización de este proyecto se han utilizado diferentes librerías de programación, principalmente librerías de PCL y un software proporcionado por OpenNI [10] que nos permita la interacción entre la cámara y nuestro programa.

La organización OpenNI (Figura 18) es una industria dirigida sin fines de lucro, destinada a promover y certificar la compatibilidad e interoperabilidad de Interacción Natural (NI) dispositivos, aplicaciones y middleware.

Figura 18: Logo OpenNI.

Como un primer paso hacia este objetivo, la organización ha puesto a disposición de un marco de código abierto - el marco OpenNI - que proporciona una interfaz de programación de aplicaciones (API) para escribir aplicaciones que utilizan la interacción natural. Esta API cubre la comunicación con ambos dispositivos de nivel bajo (por ejemplo sensores de visión y de audio), así como soluciones de alto nivel de middleware (por ejemplo, para el seguimiento visual utilizando la visión por ordenador).

Gracias a OpenNI podemos escribir algoritmos sin importar qué sensor se esté utilizando ya que ofrece a los fabricantes la posibilidad de compatibilidad e interoperación.

Actualmente los módulos soportados de sensores son:

- Sensor 3D
- Cámara RGB
- Cámara de infrarrojos
- Dispositivo de audio (un micrófono o una matriz de micrófonos)

Disponen de muchos programas desarrollados como el seguimiento de cuerpos mostrado en la captura 19 o el reconocimiento facial entre otros.

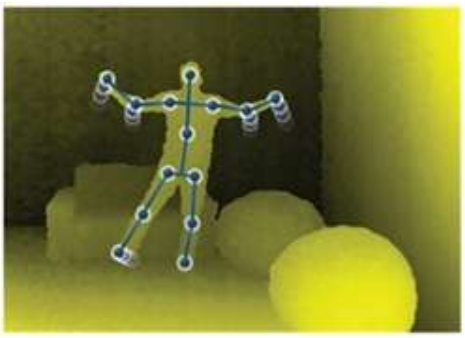


Figura 19: Programa Skeleton de OpenNI.

En la figura 20 se representa el concepto de OpenNI en la que cada una de las tres capas representa un elemento integral:

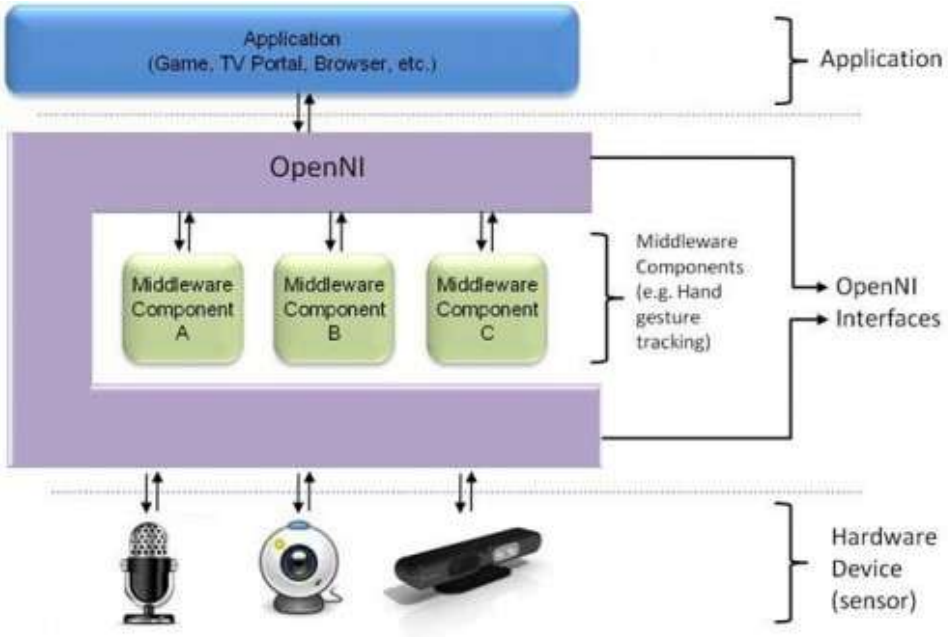


Figura 20: Representación del concepto OpenNI como enlace.

- Alto: Representa el software que implementa las aplicaciones naturales de interacción.
- Medio: Representa OpenNI, proporcionando interfaces de comunicación que interactúan tanto con los sensores y los componentes de middleware, que analizan los datos del sensor.
- Abajo: Muestra los dispositivos de hardware que capturan los elementos visuales y de audio de la escena.

4.2. Point Cloud Library

En la parte de programación nos apoyaremos en librerías PCL, Point Cloud Library [11], es a gran escala un proyecto abierto para el procesamiento de nubes de puntos en 3D. Dichas librerías contienen numerosos algoritmos entre los que podemos distinguir avanzados algoritmos de filtrado, de estimación de funciones, reconstrucciones de superficies, registro de datos, ajustes de modelos y segmentación entre otros. En la Figura 21 podemos ver el logo de PCL.

Figura 21: Logo PCL.

Estos algoritmos tienen diferentes usos como podrían ser el filtrado de valores atípicos propios del error cometido por el hardware ante la captura de imágenes de un entorno, extracción de puntos clave de una imagen, crear superficies suavizadas del entorno o llevar a cabo una disminución de carga de imágenes reduciendo el número de puntos necesarios para formar dicha imagen sin perder información entre otras finalidades.

PCL es un software de código abierto libre, además de ser gratuito para uso comercial e investigación.

PCL además es multiplataforma, ya que ha sido compilado y desplegado de manera correcta en Windows, Linux, MacOS, y Android/ iOS.

Para simplificar el desarrollo, PCL se divide en una serie de librerías de código más pequeño para conseguir una fragmentación tal que permita simplificar el desarrollo de programas. Gracias a esta modularidad se logra una facilidad de compilación dentro de aquellas plataformas que no disponen de un potencial amplio para el compilado de complejos programas.

Entre estas bibliotecas de código más pequeño podemos encontrar:

- libpcl filters: donde podemos encontrar filtros que nos permiten la reducción de resolución, eliminación de valores atípicos, extracción de índices, proyecciones, etc.
- libpcl keypoints: implementa características 3D como curvaturas normales de la superficie, estimación de un punto, descriptores FPFH (Fast Point Feature Histograms), descriptores NARF, RIFT, RSD y VFH

- libpcl io: implementa operaciones de entrada y salida de datos como la lectura de PCD así como la escritura de estos archivos.
- libpcl segmentation: implementa los métodos de extracción, modelos de ajuste en función de métodos paramétricos.
- libpcl surface: implementa técnicas de reconstrucción de superficie o mallado, representación de espacios poligonales convexos, etc.
- libpcl registration: registro de nubes de puntos tales como PCI.
- libpcl rangeimage: implementa soporte para imágenes de rango creado a partir de conjuntos de datos de nubes de puntos.

Además PCL contiene su propia librería de visualización basada en VTK. VTK ofrece un gran soporte multiplataforma para la representación de nubes de puntos 3D.

La librería de visualización PCL tiene la intención de integrar PCL con VTK, proporcionando una capa de visualización integral de estructuras tridimensionales formadas por nubes de puntos. Su propósito principal es el poder crear prototipos de forma rápida y visualizar algoritmos de manera rápida y sencilla. Esta librería de visualización nos ofrece:

- Métodos para el establecimiento de propiedades de puntos (Colores, Tamaños de punto, opacidad, etc.)
- Métodos básicos para la realización de figuras 3D en la pantalla como cilindros, esferas, líneas, polígonos, etc.
- Módulos de visualización de histogramas de gráficos 2D.

Nosotros trabajaremos íntegramente analizando archivos .pcd, formato utilizado dentro de las librerías point cloud. Este formato tiene intención de complementar los formatos existentes de nubes de puntos para llevar a cabo una mejor compatibilidad con PCL.

Cada archivo pcd (Figura 22) tiene una cabecera que identifica y declara ciertas propiedades de los datos de la nube de puntos contenido dentro de ese archivo. Dicho encabezado debe ser codificado en ASCII y contendrá la siguiente información.

1. Versión: donde se especifica la versión del archivo .pcd.
2. Campos: especifica el campo que cada punto debe contener. (XYZ, XYZRGBA…)
3. Tamaño: especifica el tamaño de cada dimensión.
4. Tipo: especifica el tipo de cada dimensión.
5. Count: especifica cuantos elementos, tales como descriptores, tiene cada dimensión.
6. Ancho: especifica el ancho del conjunto de datos.
7. Alto: especifica el alto del conjunto de datos.
8. Punto de toma de datos: especifica el punto de vista de adquisición del conjunto de datos.
9. Número de puntos: especifica el número de puntos contenidos en la nube.

Las ventajas de este tipo de formato son principalmente su flexibilidad, su velocidad y la robustez que le da ser el archivo nativo de PCL además de:

- La capacidad de almacenar y procesar conjuntos de datos organizados de nubes de puntos. Esto es de suma importancia para aplicaciones en tiempo real y áreas de investigación como la robótica, realidad aumentada, etc.
- Para binarios mmap / munmap tiene la forma mas rápida de cargar y guardar datos en el disco.
- El almacenamiento de diferentes tipos de datos (todos los primitivos son soportados) permite que los datos de las nubes de puntos sean mas flexibles y eficientes en lo que respecta a procesamiento y almacenamiento. Las dimensiones de puntos no validos se almacenan como NaN.
- Histogramas ND de descriptores de funciones (muy importantes para las aplicaciones de visión 3d de percepción.

A pesar de que PCD es el formato nativo de PCL, la biblioteca pcl_io ofrece la posibilidad de guardar y cargar archivos de otros formatos.

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
0.97192 0.278 0 4.2108e+06
0.944 0.29474 0 4.2108e+06
0.98111 0.24247 0 4.2108e+06
0.93655 0.26143 0 4.2108e+06
0.91631 0.27442 0 4.2108e+06
0.81921 0.29315 0 4.2108e+06
0.90701 0.24109 0 4.2108e+06
0.83239 0.23398 0 4.2108e+06
0.99185 0.2116 0 4.2108e+06
```

Figura 22: Ejemplo de archivo en formato .PCD.

4.3. C++ y la programación orientada a objetos

En la realización de este programa utilizaremos el lenguaje de programación C++ [12,13], diseñado en 1980 por Bjarne Stroustrup. La primera finalidad por la que se creó fue la extensión del exitoso lenguaje C para poder manipular objetos.

La programación orientada a objetos se creo buscando una programación estructurada cuya idea principal es separar las partes complejas de un programa en diferentes segmentos que sean ejecutados en función de las necesidades. Esto daría a una actividad modular diferente a la programación que generaba grandes cantidades de código en un solo bloque de manera lineal.

A diferencia de C, el cual se rige principalmente mediante funciones que se llaman unas a otras entre las que podemos intercambiar información, con C++ disponemos más versatilidad gracias a la existencia de clases. Dentro de estas clases podremos definir las propiedades y el comportamiento de un objeto en concreto. Los objetos son instancias de clases. Por ejemplo la clase perro define los atributos (número de patas, forma del hocico, raza, etc.) y las funciones (ladrar, correr, buscar, etc.) de dicha clase. Luego podremos tener un objeto de esa clase como podría ser un perro Milú concreto, que tenga unos atributos determinados.

Durante el desarrollo del programa utilizaremos numerosos objetos contenidos dentro de las bibliotecas PCL.

4.4. Clases y funciones empleadas

En este apartado veremos las clases de pcl (Sección 4.2) empleadas en el programa que son:

- PCDReader.

Gracias a esta clase podremos leer nubes de puntos de entrada. La única función de la clase PCDReader llamada será:

1. read ("nube a evaluar", *nubedememoria);

En esta función cargaremos la nube a evaluar en una nube de memoria desde la que trabajaremos posteriormente.

- Passthrought.

Gracias a esta clase podremos filtrar nubes de puntos en X, Y o Z. Las funciones de la clase utilizadas serán:

1. setInputCloud (nubetratada);

Con esta función introducimos la nube que queremos filtrar.

2. setFilterFieldName ("X Y o Z");

En función de que pongamos x, y o z filtraremos en un eje u otro.

3. setFilterLimits (a,b);

Colocamos los límites de la imagen con los que queremos quedarnos en el eje elegido anteriormente.

4. filter (*nube segmentada);

Definimos la nube de destino donde queremos guardar el resultado de la operación.

- NormalEstimation.

Será utilizada para estimar las normales de los puntos de la nube.

1. setInputCloud (nubetratada);

Con esta función introducimos la nube de la que queremos extraer las normales.

2. setSearchMethod ("metodo");

Con esta función estableceremos el método de búsqueda de los vecinos más cercanos.

3. setKSearch (número);

Colocamos el número de vecinos que queremos encontrar, en nuestro caso con búsqueda de arboles kd.

4. compute (*nube de normales);

Definimos la nube de destino donde queremos guardar las normales obtenidas.

- SACSegmentationFromNormals.

Será utilizada para estimar las normales de los puntos de la nube.

1. setModelType(modelo buscado);

Con esta función introducimos el modelo al que buscamos que los puntos se ajusten.

2. setMaxIterations(número de iteraciones);

Definimos el número de iteraciones que realizaremos hasta dar un resultado.

3. setDistanceThreshold (0-1);

Con un número decimal de 0 a 1 definimos el grado de ajuste que buscamos sobre el modelo buscado.

4. setAxis(vector);

Definimos el vector que buscamos como definir como perpendicular al plano buscado.

5. setEpsAngle (Angulo en radianes);

Establecemos un margen de error en gradianes sobre el vector buscado.

6. segment (*inliers, *coeficientes);

Definimos los archivos donde almacenar los inliers y los coeficientes obtenidos.

- ExtractIndices.

Será utilizada para extraer los inliers resultantes de la utilización del algoritmo RANSAC sobre la nube.

1. setInputCloud (nube);

Introducimos la nube de la que queremos extraer los inliers obtenidos del modelo.

2. setIndices (inliers);

Definimos los inliers con los que trabajar.

3. setNegative (booleano);

Booleano para obtener únicamente los inliers (false) o la parte que no esta definida como inliers dentro de la nube (true).

4. filter (*nube resultante);

Establecemos la nube donde guardar el resultado de la operación.

- ProjectInliers.

Gracias a esta función proyectaremos los puntos para trabajar sobre un plano unificado.

1. setInputCloud (nube);

Introducimos la nube que queremos proyectar.

2. setModelType (modelo);

Definimos el modelo que queremos proyectar, en nuestro caso SACMODEL_PLANE.

3. setModelCoefficients (coeficientes);

Coeficientes del plano dentro de la ecuación general del tipo $Ax + By + Cz + D = 0$.

4. filter (*nube resultante);

Establecemos la nube donde guardar el resultado de la operación.

- SegmentDifferences.

Con esta función restaremos las nubes de puntos correspondientes al suelo y a los objetos para poder tener la nube de puntos final. Utilizaremos las siguientes funciones:

1. setInputCloud (nube);

Introducimos la nube que inicial sobre la que restaremos las barreras.

2. setDistanceThreshold (0-1);

Establecemos un número decimal de 0 a 1 donde 0 será la resta exacta y a medida que el número sea mas grande se creara un umbral de error sobre esa resta.

3. setTargetCloud (nube compuesta por las barreras);

Nube compuesta por las barreras que restaremos al suelo.

4. segment (*nube resultante);

Establecemos la nube donde guardar el resultado de la operación.

- PCDWriter.

Gracias a esta clase podremos guardar nubes de puntos obtenidas tras la realización de alguna de las anteriores operaciones. Utilizaremos una única función dentro de esta clase.

1. write (".pcd destino", *nube a guardar);

En esta función establecemos la nube que queremos guardar y el archivo pcd que queremos generar donde se almacenará la nube seleccionada.

5. Estimación de zonas cruzables

A continuación llevaremos a cabo una descripción detallada de los diferentes pasos realizados para llevar a cabo el programa buscado. Realizaremos unas valoraciones previas (Sección 5.1) y un diagrama de flujo explicativo (Sección 5.2) para situarnos en las condiciones de trabajo. Pasaremos después a explicar cada una de las acciones de nuestro programa (Secciones 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9 y 5.10).

5.1. Valoraciones previas

Previamente a la realización de cualquier función del proyecto deberemos evaluar las condiciones iniciales desde las que partimos. La principal finalidad de nuestro proyecto es dotar a un robot de un sistema que suministre nubes de puntos del plano cruzable sobre el que puede transitar.

Inicialmente consideraremos un robot de 38 cm de alto sobre el que colocaremos nuestro sensor Kinect (Sección 3.3) siendo este el punto más alto del conjunto robótico. La altura del punto de visión del sensor será de 43 cm (38 cm del robot y 5 desde la base de la Kinect hasta la salida de visión). La altura total del conjunto será de 45 cm (38 cm del robot a la que sumamos 7cm de la altura total del sensor Kinect). La altura del robot es una altura tomada de manera aleatoria, se podrá adaptar a diferentes alturas dependiendo del robot utilizado.

Nuestra cámara dispone de 43º de visión vertical y un movimiento vertical de ±27º de la cámara dividido en diez posiciones. Si colocamos la cámara paralela al suelo tendremos la situación descrita en la Figura 23:

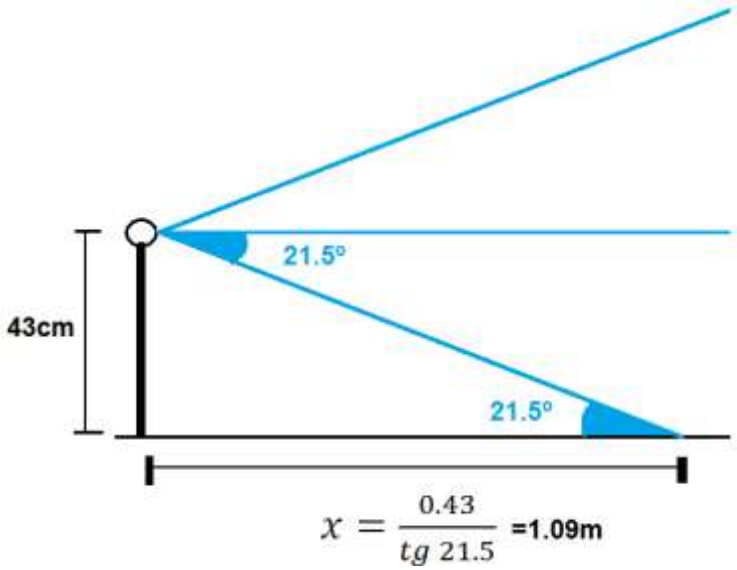


Figura 23: Rango de visión del sensor Kinect.

Con el que podemos determinar que el suelo lo podríamos comenzar a percibir a una distancia de 1.09m. Las especificaciones técnicas determinan que el rango de visión mínimo de la Kinect es de 1.2m aunque en condiciones favorables de visión podemos captar objetos que disten 1m del foco.

Por ello, aprovechando el movimiento vertical, inclinamos 5.4º el sensor hacia el suelo para poder capturarlo a una menor distancia obteniendo el resultado representado en la Figura 24:

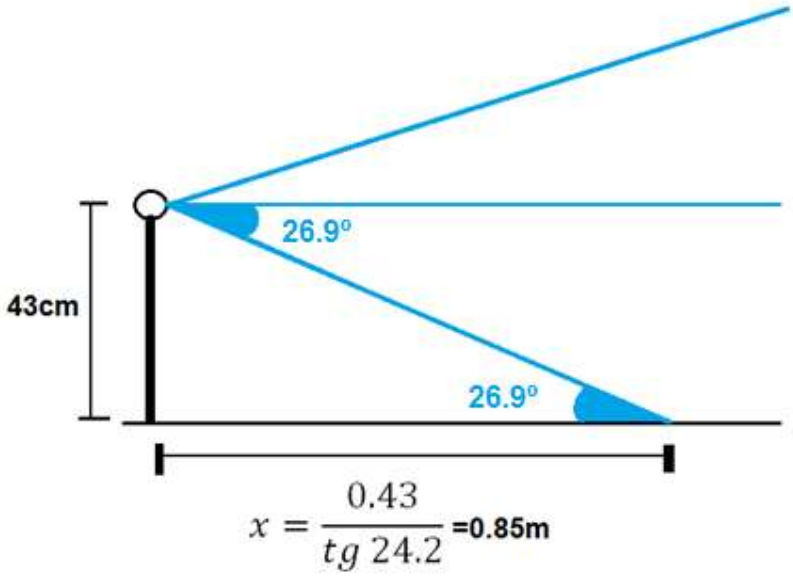


Figura 24: Nuevo rango de visión con inclinación.1

De esta forma podremos capturar información a una distancia menor de nuestro robot. Los ejes de visión estarán establecidos a partir de ahora de la manera expuesta en la Figura 25:

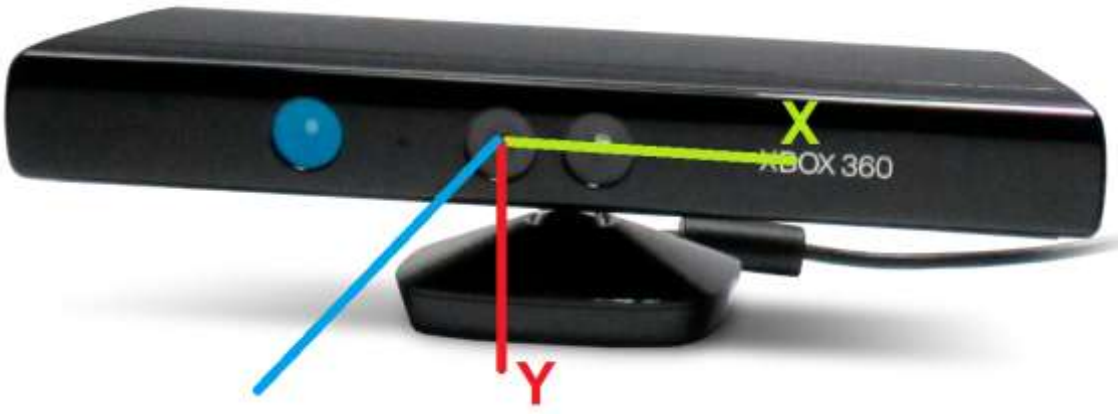
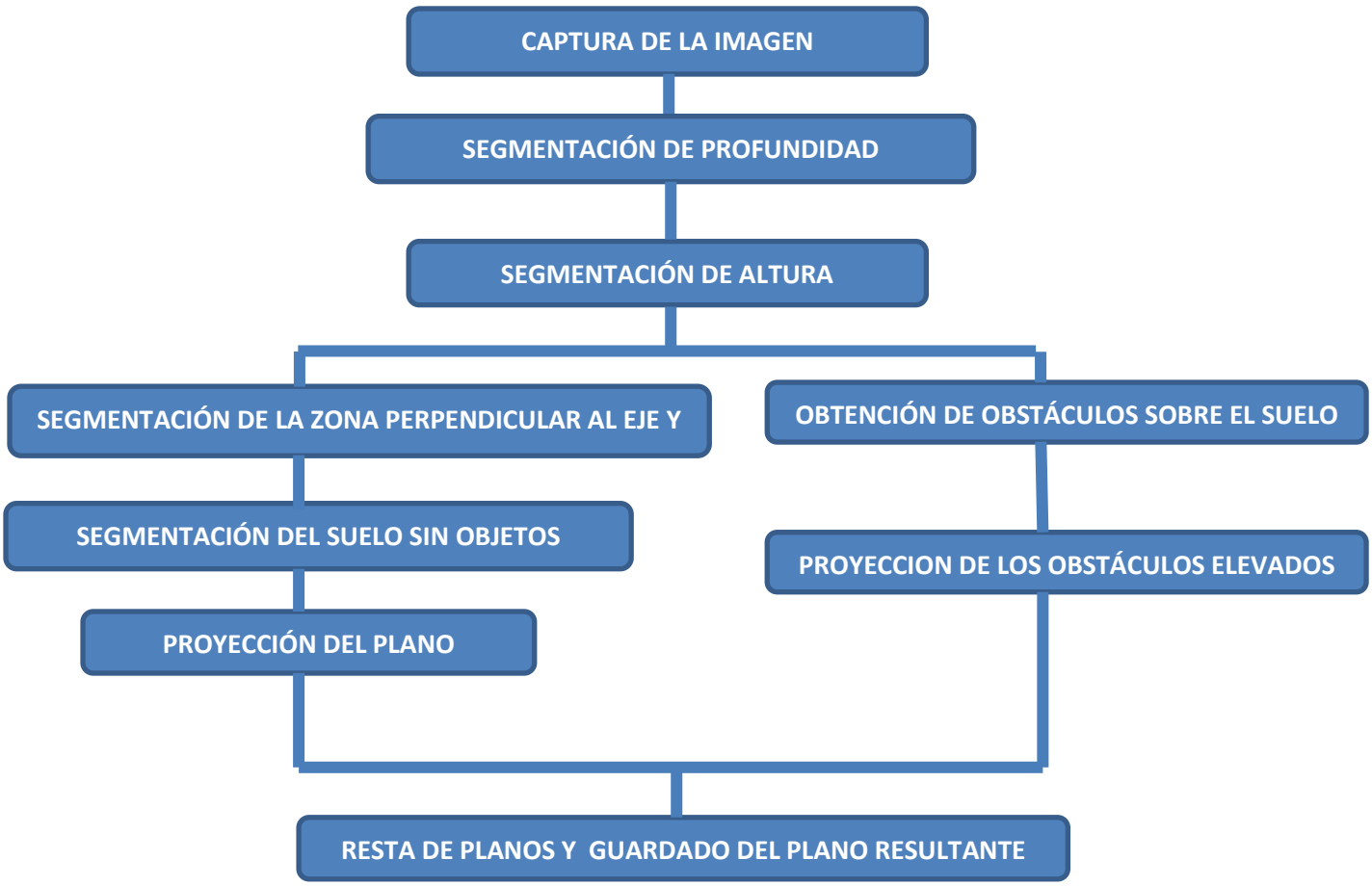


Figura 25: Sistema de coordenadas del sensor.

5.2. Diagrama de flujo del proceso



Como podemos observar tras la captura de la imagen inicial, segmentaremos ésta para reducir el área de interés (Secciones 5.3 y 5.4). Luego extraeremos un plano perpendicular al eje Y con un margen (Sección 5.5) y del resultado segmentaremos el plano del suelo sin objetos total (Sección 5.6). Realizaremos un procedimiento similar pero obteniendo los obstáculos sobre el suelo (Sección 5.8). Proyectaremos las dos nubes resultantes para trabajar sobre la misma superficie (Secciones 5.7 y 5.9). Finalmente restaremos los planos para tener una superficie final íntegramente transitable y guardaremos el resultado (Sección 5.10).

5.3. Segmentación de profundidad

La toma de imágenes de la Kinect nos dará una nube de puntos en función del rango de visión disponible y la estancia en la que se encuentre. Debido al amplio error que puede cometer al obtener puntos a más de dos metros y medio, segmentaremos hasta esa distancia en el eje Z la nube de puntos.

Cargaremos la imagen utilizando la función read descrita en la Sección 4.5 y posteriormente utilizaremos un filtro PassThrough también descrito en esta sección. El funcionamiento de este filtro es sencillo, se realiza un barrido de los valores de todos los puntos pertenecientes a la imagen y todos aquellos que superen el valor impuesto como condición se eliminan. En la Figura 26 vemos como una imagen en la que se captura una mesa, una taza y la pared de fondo se reduce al punto de interés que es la mesa con la taza.

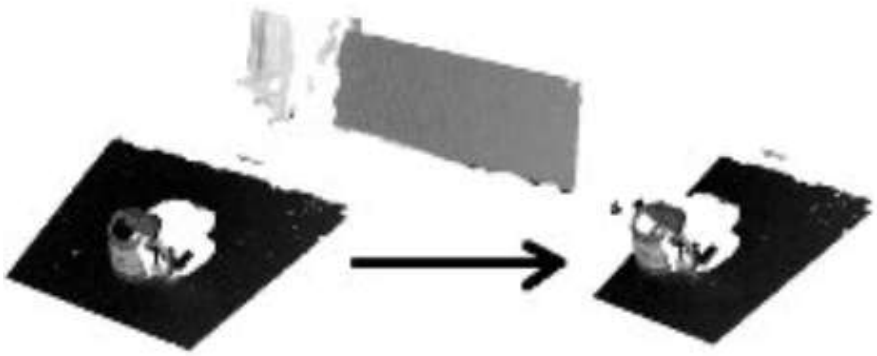


Figura 26: Ejemplo de uso del filtrado PassThrought.

Como utilizaremos el eje de coordenadas que utiliza el sistema de la Kinect (Figura 25) deberemos segmentar una distancia de 2.5m(a la que añadiremos 15cm del error posible tomado de la Figura 12) desde el punto de origen. $[\,0\,,2'65\,]m$.

Para utilizar este filtro primero creamos el objeto passz de la clase passthrough:

- pcl::PassThrough<pcl::PointXYZRGBA> passz

Cargamos la imagen sobre la que queremos trabajar (en nuestro caso la nube es inicial):

- passz.setInputCloud (inicial);

Establecemos el eje en el que filtraremos y los rangos de este filtrado (de 0 a 2.65m).

- passz.setFilterFieldName ("z");
- passz.setFilterLimits (0, 2.65);

Por último guardamos la imagen resultante en una nueva nube de puntos para pasar a los siguientes pasos del procedimiento.

En las siguientes imágenes veremos una nube genérica tomada con la Kinect antes y después de realizar el filtrado. La Figura 27 presenta la vista de pájaro de una estancia con un pasillo tomada desde la izquierda de la imagen.

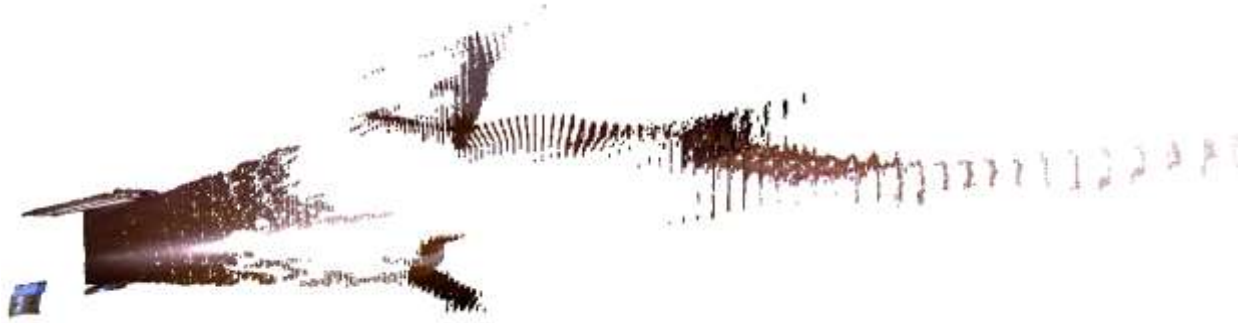


Figura 27: Vista de pájaro previa segmentación en profundidad.

Y una vez realizamos el filtrado de profundidad tenemos una estancia nueva mostrada en la Figura 28:

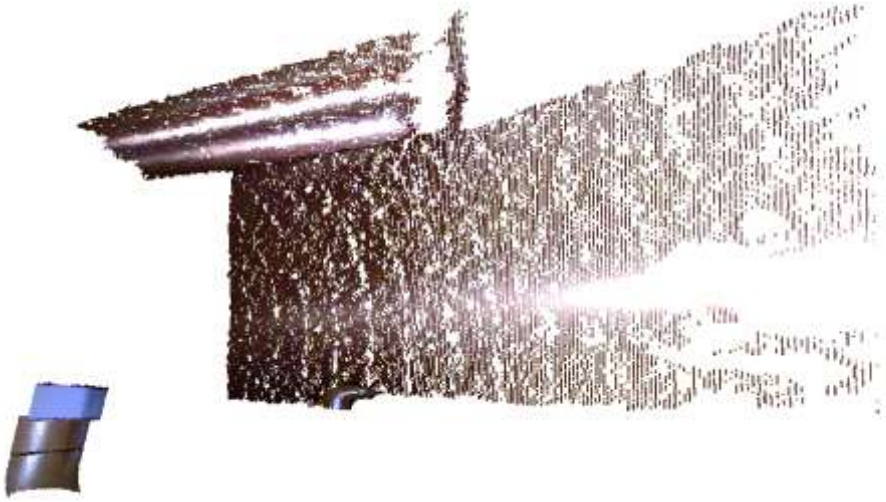


Figura 28: Resultado de la segmentación en profundidad.

Donde podemos observar como se ha segmentado todos los puntos a una distancia mayor a 2.65m que corresponden a la parte derecha de la imagen.

5.4. Segmentación de altura

Dada la situación de nuestro problema, los puntos que queden por encima del robot son exceso de información. Consideraremos sobrantes por lo tanto aquellos puntos que superen los 45cm del robot (43 hasta el foco mas 2 hasta el puto mas alto), además de un error de 15cm (tomado de la grafica de errores mostrada en la Figura 12).

Hemos de tener en cuenta la inclinación de la cámara Kinect (Figura 29) a la hora de aplicar el rango del filtrado ya que podría eliminar información necesaria cuanto más nos distanciemos del foco. Para compensar el grado de inclinación llevaremos a cabo el siguiente cálculo:

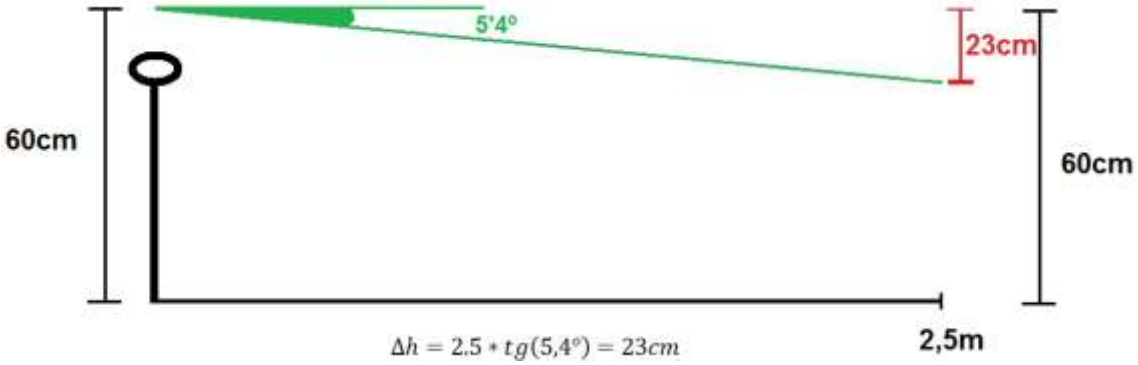


Figura 29: Cálculo de la variación de altura debida a la inclinación.

Deberemos de segmentar a una altura de 17cm (15 del error mas 2 hasta la parte superior) mas 23cm para compensar la inclinación. Así quedará únicamente la información necesaria como la mostrada en la Figura 30, que nos afectaría dentro del rango que evaluaremos de 2.5 metros de distancia previamente segmentado.

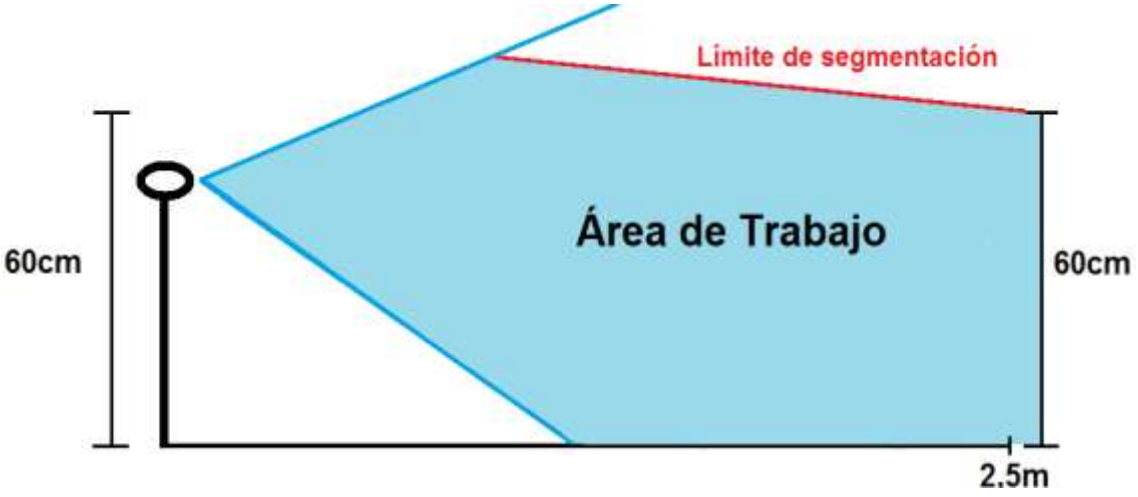


Figura 30: Representación del área de trabajo final.

El procedimiento es similar a la segmentación en profundidad salvo por que deberemos cambiar el eje de segmentación así como los limites de esta.

- passy.setFilterFieldName ("y");
- passy.setFilterLimits (-0.30,0.43);

Realizando esto sobre la nube 31 sin transformaciones como la siguiente.

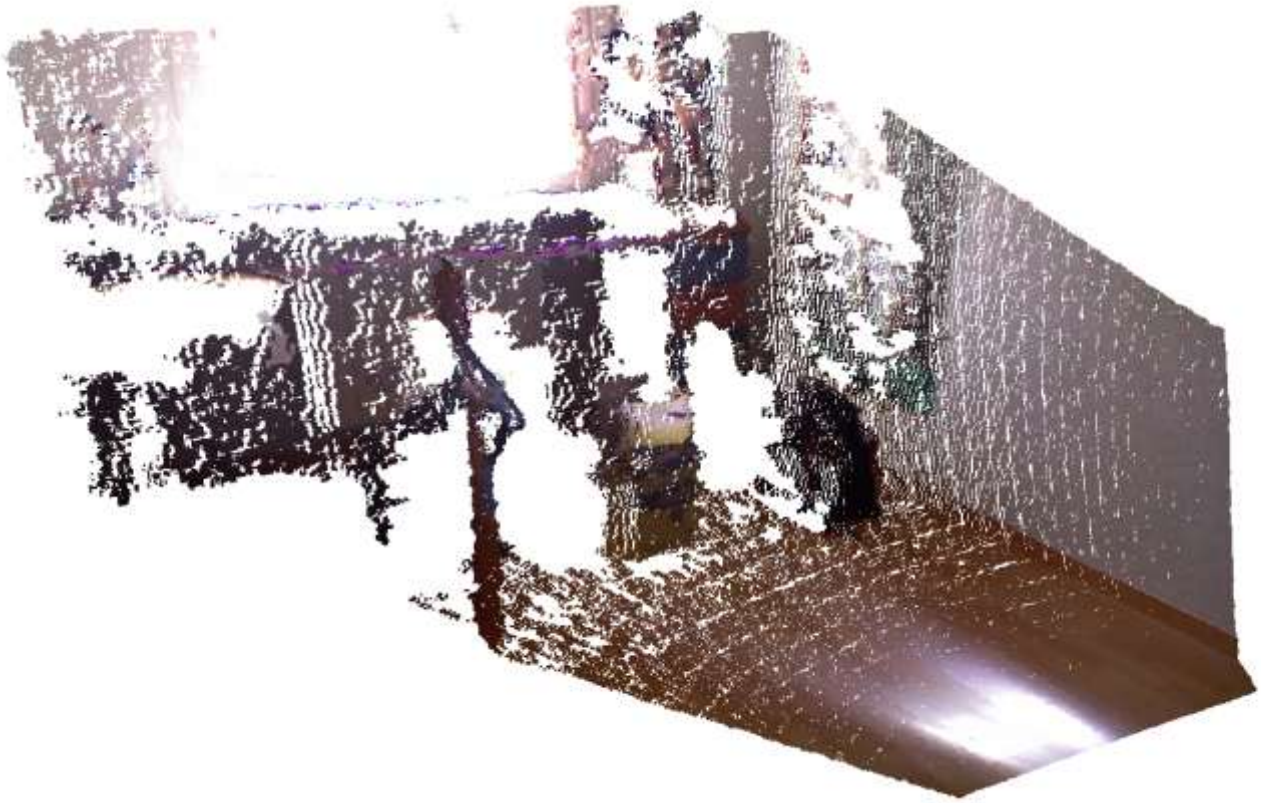


Figura 31: Imagen previa segmentación en altura y profundidad.

Realizamos el filtrado definido previamente por lo que obtendremos el siguiente resultado:

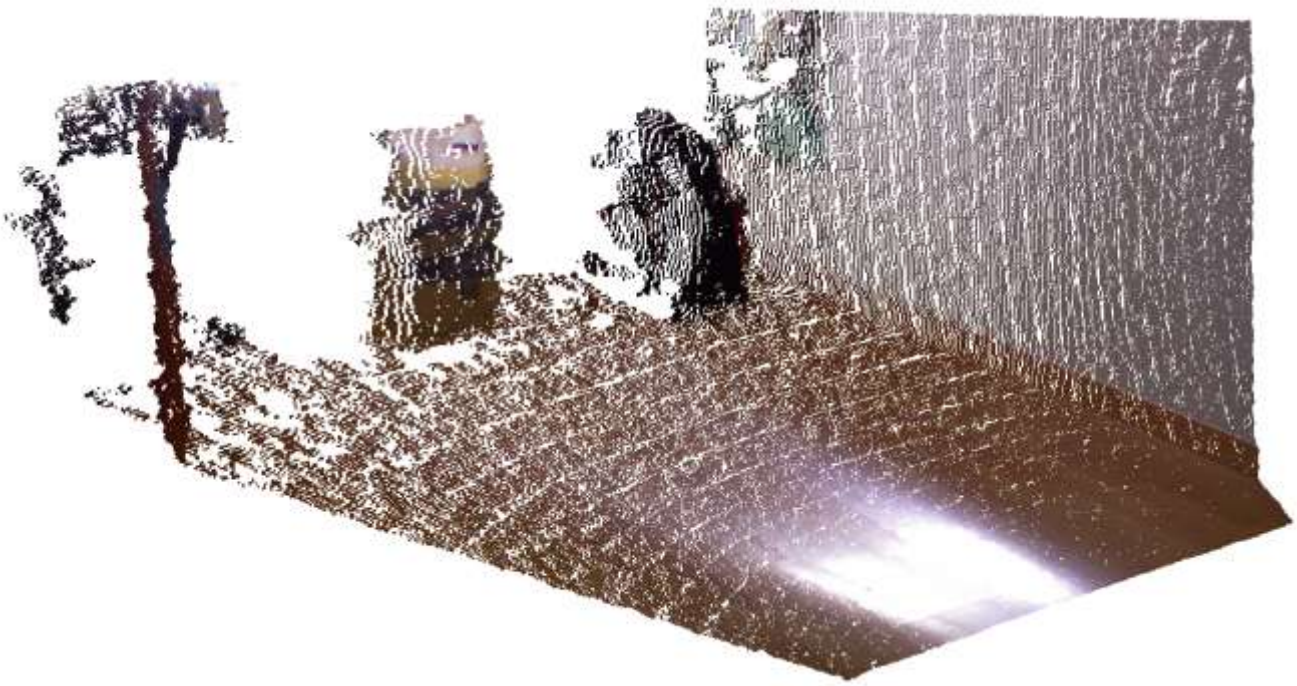


Figura 32: Imagen segmentada en altura y profundidad.

Podemos observar en la Figura 32 como todos los puntos superiores son eliminados obteniendo una imagen menos pesada ya que se ha reducido considerablemente el número de información irrelevante.

5.5. Segmentación de la zona perpendicular al eje Y

Una vez tenemos simplificada la imagen, el siguiente paso que nos interesa es obtener el plano del suelo.

Podríamos utilizar un método que extrajese el plano de la foto pero podemos dar con el caso en el que nos encontremos enfrente de una pared o un objeto plano de gran tamaño. Ésto nos llevaría a posibles errores en el programa ya que tomaría el plano que mayor número de puntos aportase pudiendo ser este el erróneo.

Para llevar a cabo este proceso utilizaremos un método llamado KdTree para buscar los vecinos más cercanos de los cuales podremos estimar las normales y posteriormente con un Algoritmo RANSAC para buscar el área de interés.

A continuación explicaremos el método KdTree, el algoritmo RANSAC y por último la aplicación practica de dichas herramientas.

- # Estimación de normales mediante KdTree

Trabajar con nubes de puntos que contienen tanta cantidad de información es una tarea ardua por lo que se utilizan de manera usual estructuras tipo árbol que permitan una reducción del tiempo de trabajo acelerando la búsqueda. En nuestro trabajo deberemos de buscar normales pero para ello se han de explorar los vecinos cercanos al punto que estamos evaluando. Para ello utilizaremos el KdTree [11, 14, 15], un árbol clásico que se utiliza para la búsqueda del vecino más cercano a través de nodos que acumulan información. Es un caso especial de los árboles BSP (Binary space partitioning) en el que se subdivide recursivamente un espacio en elementos convexos creando hiperplanos. Este particionado es arbitrario mientras que el particionado del KdTree siempre es perpendicular, de manera equilibrada al buscar que el prototipo buscado tenga la misma probabilidad de estar en las distintas subdivisiones.

Para obtener la normal de un punto deberemos tener en cuenta la información de los puntos vecinos a él. Mediante el algoritmo KdTree, utilizaremos aproximaciones para llevar a cabo esta búsqueda de vecindad.

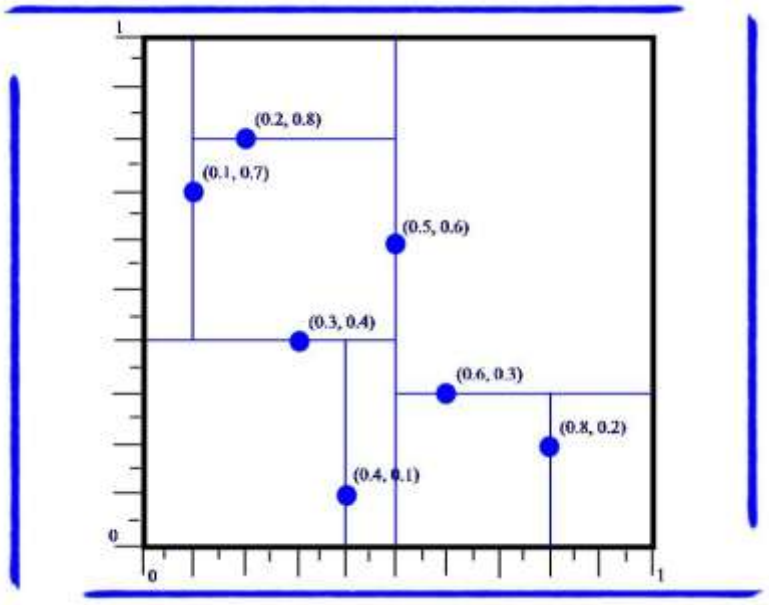


Figura 33 Ejemplo de funcionamiento del algoritmo KdTree

Un ejemplo gráfico del funcionamiento del KdTree lo podemos encontrar en la Figura 33 donde cada nodo hace una partición del plano en dos "subplanos" mediante una recta perpendicular al eje. Dicha recta esta determinada por la componente que indica el discriminante buscado.

En nuestro programa utilizaremos este método utilizando un objeto de la clase NormalEstimation (Sección 4.4). Introduciremos el método de búsqueda Kdtree mediante la función setSearchMethod y el número de vecinos que queramos buscar utilizando setKSearch. Esto proporcionará una superficie a partir de la cual se generará una matriz de covarianza. Se analizará posteriormente los vectores y valores propios de dicha matriz para llevar a cabo la estimación de la normal del punto perteneciente a dicha superficie.

Gracias a esta clase perteneciente a PCL podremos obtener las normales de cualquier objeto como podemos ver en la Figura 34.

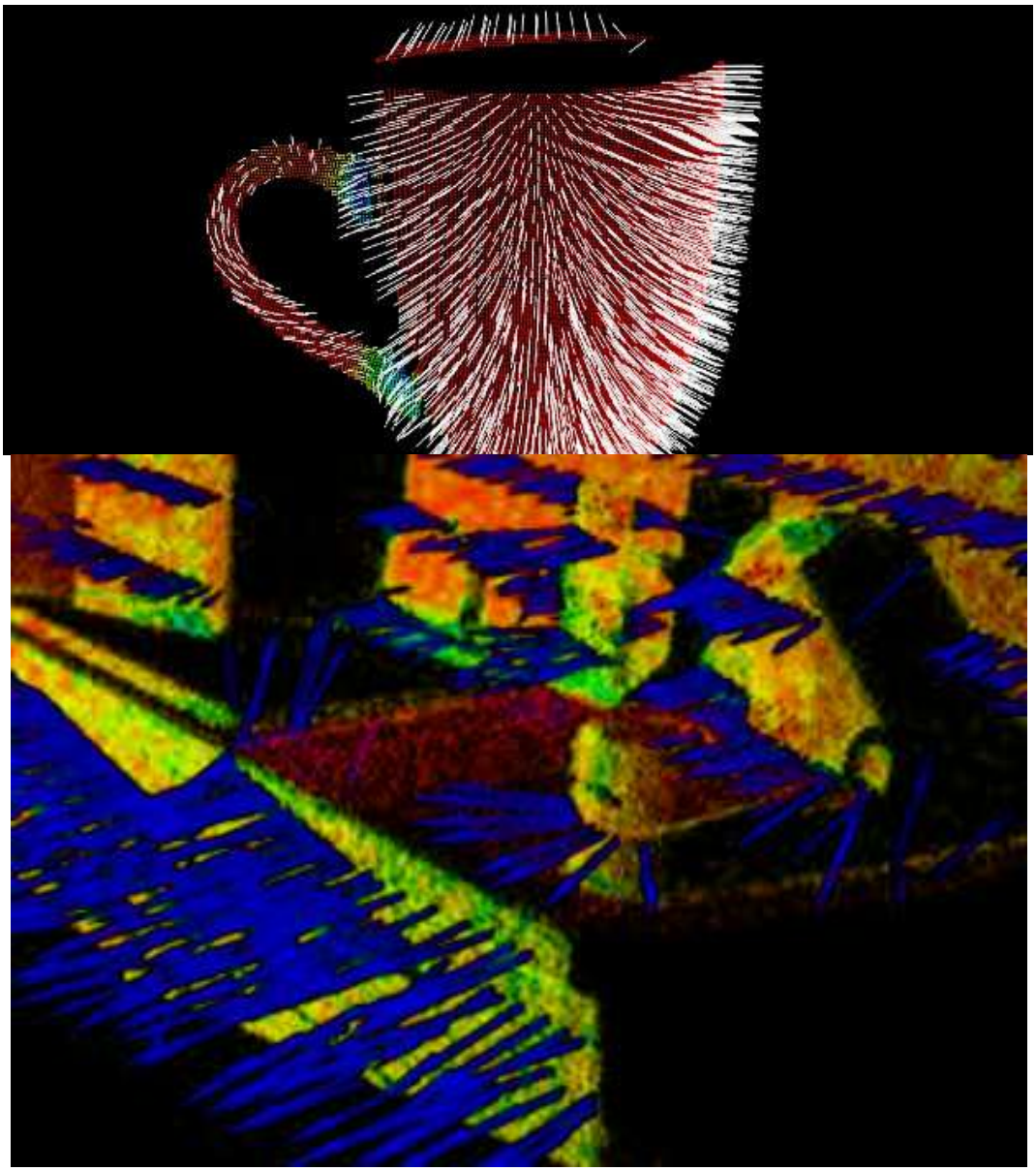


Figura 34: Imagenes con las normales representadas.

- **Random Sample Consensus Model (RANSAC)**

Random Sample Consensus (RANSAC) [11], en español Conjunto de Consenso de Muestras Aleatorias, es un algoritmo que a partir de un conjunto de datos observados es capaz de estimar los parámetros de un modelo matemático. Fue desarrollado por Fischer y Bolles [16] para encontrar los parámetros de un modelo que mejor se ajuste a un conjunto de datos dados.

Como principal ventaja tiene la gran robustez que presenta para estimar modelos matemáticos con un alto grado de precisión a pesar de tener una gran proporción de datos atípicos. Además a mayor número de iteraciones mayor porcentaje de acierto.

Entre sus desventajas está la inexistencia de límite superior de tiempo que se necesita para calcular los parámetros. Además solo se puede estimar un modelo para un conjunto de datos. Si queremos estimar un conjunto de datos a varios modelos deberemos de realizar tantas pruebas como modelos busquemos (en este caso sería interesante valorar la transformada de Hough [17]). Su mayor problema es que no se puede establecer un límite superior en el tiempo de ejecución.

El funcionamiento del algoritmo RANSAC se basa en la selección iterativa y aleatoria de subconjuntos de datos dentro del conjunto total. Dentro de ese subconjunto total podremos diferenciar entre los "inliers" (datos cuya distribución se explica por un modelo determinado) y los "datos atípicos" (datos que no encajan en el modelo). A partir de los puntos tomados se estiman los parámetros que definen el modelo buscado y se evalúa con que grado se ajustan esos datos al modelo. En caso de haber suficientes datos del subconjunto que se ajustan correctamente a dicho modelo, nos encontramos ante una correcta representación. Este algoritmo se repite iterativamente.

Dentro de nuestro contexto de trabajo nosotros tendremos una imagen de la cuál tomaremos un subconjunto aleatorio. Dentro de esa imagen tendremos una serie de inliers hipotéticos que cumplirán el modelo buscado. Todos los demás datos sobrantes los probamos frente al modelo de inliers hipotéticos buscando un ajuste menos estricto. Si ese punto se ajusta correctamente al formado por el conjunto de inliers hipotéticos pasa a considerarse parte de ese conjunto. Con el nuevo modelo de inliers hipotéticos se repite el mismo proceso. El modelo estimado pasa a considerarse como correcto si contiene gran cantidad de inliers hipotéticos siendo finalmente evaluado mediante la estimación del error de los inliers relativos al modelo.

El conjunto de entradas necesarias es

- **Imagen:** Conjunto de datos de imagen a analizar.
- **Modelo**: Modelo que buscamos ajustar.
- **n:** Número mínimo de muestras necesarias para estimar los parámetros del modelo.
- **k:** Número máximo de iteraciones.
- **t:** Valor umbral o distancia de referencia, parámetro usado para determinar cuando un dato forma parte del modelo buscado.
- **d:** Número de valores necesarios para afirmar que el modelo se ajusta correctamente.

Dando como resultado una serie de salidas:

- **Best_model:** Parámetros del modelo que mejor se ajusta.
- **Best_Consensus_set:** Error relativo de los datos respecto a dicho modelo
- **Best_ error:** Datos de entrada que mejor se han ajustado al modelo.

En nuestro caso tomaremos una nube de puntos aleatoria con 3000 puntos como la de la Figura 35:

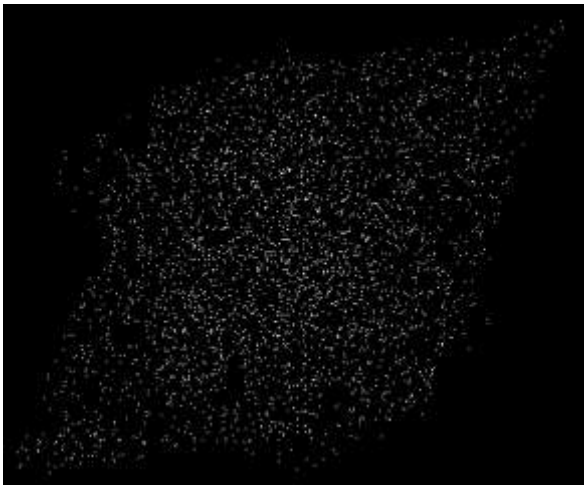


Figura 35: Nube de puntos aleatoria

A continuación mediante una búsqueda RANSAC podremos obtener distintos modelos:

- Esfera mostrada en la Figura 36 formada por 1049 puntos tras 1000 iteraciones y un umbral de 0.01:

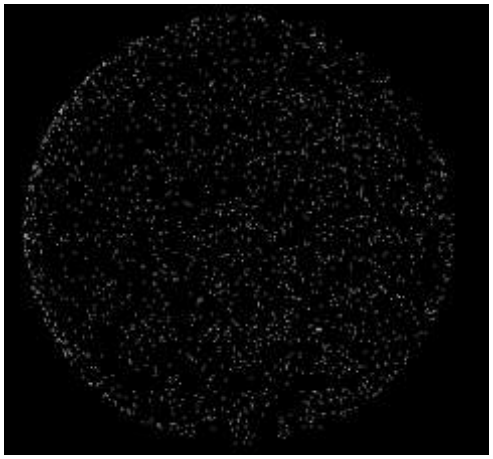


Figura 36: Resultado de la búsqueda del modelo esférico.

- O el modelo buscado en esta mismo capítulo, como es el modelo planar mostrado en la Figura 37:

Figura 37: Resultado de la búsqueda del modelo planar.

Obtenido tras 1000 iteraciones y con un umbral de 0.01 tenemos 1706 puntos que se ajustan a nuestro modelo buscado.

- # Aplicación en el programa

En el programa primero buscaremos las normales de nuestra nube de puntos y posteriormente buscaremos un plano perpendicular al eje Y. Esto nos proporcionará una serie de inliers con los que nos quedaremos posteriormente.

Primero haremos un análisis de los vecinos mediante un método Kdtree para poder acelerar la búsqueda de las normales mediante la función NormalEstimation (Sección 4.4). A continuación se presentan las funciones principales utilizadas.

Declaramos un objeto del tipo NormalEstimation llamado **ne** e introducimos los parámetros y opciones para buscar mediante un método KdTree las normales con el análisis del árbol buscando 50 vecinos próximos.

- ne.setSearchMethod (tree);
- ne.setInputCloud (YIMAGEN);
- ne.setKSearch (50);

Utilizamos 50 vecinos para tener información suficiente, ya que si utilizamos un número mayor en las zonas más lejanas a la cámara los vecinos están más separados y las zonas de estudio son mucho mayores, además del incremento de los tiempos de trabajo. Si utilizamos un número menor de vecinos los conjuntos evaluados son bastante pequeños y en zonas cercanas con mucha información da resultados erróneos.

Posteriormente utilizaremos un objeto seg de la clase SACSegmentationFromNormals (Sección 4.4) con el que, utilizando las normales obtenidas anteriormente, buscaremos la zona de interés.

- seg.setModelType (pcl::SACMODEL_PERPENDICULAR_PLANE);
- seg.setMaxIterations(1000);
- seg.setDistanceThreshold (0.05);
- seg.setAxis(Eigen::Vector3f(0,1,0));
- seg.setEpsAngle (0.2);

Definimos el vector Y (0, 1, 0) que ha de ser perpendicular al plano buscado. Introducimos una variación angular de 0.2 radianes para que el plano del suelo entre dentro del rango, ya que como especificamos en el apartado 5.1, la cámara esta inclinada. Además utilizaremos 1000 iteraciones para tener un resultado fiable y un valor de umbral que marca la fiabilidad del modelo de 0.05. Todo ello nos dará unos inliers y unos coeficientes que extraeremos guardándolos para su posterior utilización.

Se ha empleado 0.2 radianes (11.42º) para sacar un plano con un gran margen ya que a distancias mayores tenemos mucho error en la medida de los puntos. Esto provocará

que en vez de los puntos contenidos dentro de un plano, obtengamos los puntos contenidos en un prisma.

Por último, mediante un objeto ExtractIndices (Sección 4.4), obtendremos los inliers de la imagen original para utilizar únicamente la zona del suelo buscada y trabajar con ella en posteriores pasos. Podemos ver un caso práctico con una imagen inicial dada (Figura 38) pasando al resultado final obtenido (Figura 39).

Dada una imagen con los puntos de interés dados tras la segmentación en altura y profundidad:

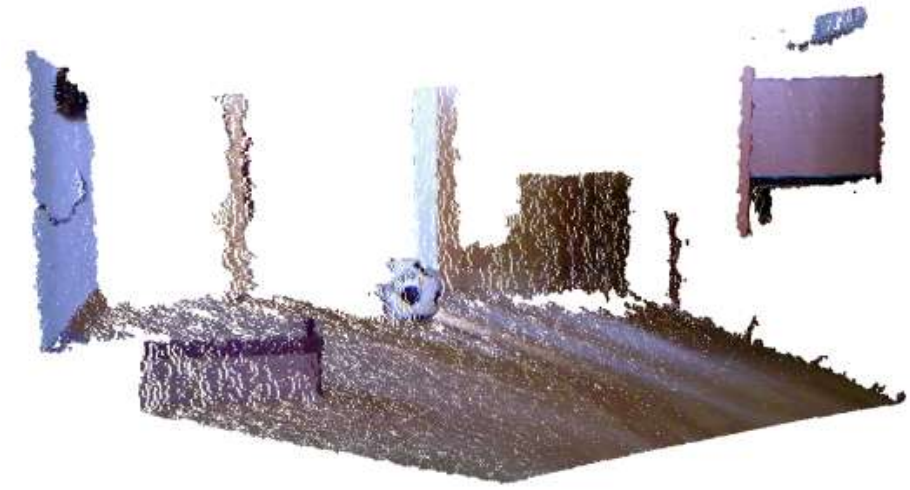


Figura 38: Imagen previa segmentación de la zona perpendicular a Y.

Al realizar el método expuesto anteriormente para la obtención de la zona perpendicular al eje Y que corresponda al suelo tenemos:

Figura 39: Resultado de la segmentación de la zona perpendicular a Y.

Tenemos información correspondiente al plano del suelo y objetos que forman parte del margen de segmentación prismático.

5.6. Segmentación del suelo sin objetos

Para corregir el error obtenido a la salida de la extracción del prisma perpendicular al plano Y, segmentaremos nuevamente la nube obtenida. Esta vez buscaremos el plano sin margen de error ya que nos encontramos con una nube de puntos muy acotada y aproximada a un plano donde no tendremos el riesgo de encontrar superficies mayores al del propio suelo, dando un resultado fiable.

Utilizaremos el mismo método del apartado anterior (Sección 5.5) con un KdTree para obtener las normales que nos ayudarán a estimar los inliers pertenecientes al suelo con el que nos quedaremos finalmente. La única diferencia es que ahora no buscaremos un plano que cumpla unas condiciones sino el plano más grande.

Si aplicamos esta herramienta a la imagen resultante de la Figura 39 obtendremos el resultado expuesto en la Figura 40:

Figura 40: Resultado de la búsqueda del plano.

5.7. Proyección del plano

El resultado de la segmentación del suelo es un plano con imperfecciones y rugosidad propia del error de medición. Para eliminar ésto y tener todo sobre un mismo plano perfecto proyectaremos todo sobre un plano que estableceremos mediante unos coeficientes:

- coefficients->values.resize (4);
- coefficients->values[0] = coefficients->values[2] = coefficients->values[3] = 0;
- coefficients->values[1] = 1.0;

Dichos coeficientes se corresponden con **A B C** y **D** pertenecientes a la ecuación general de un plano $Ax + By + Cz + D = 0$ por lo que proyectaremos todos los puntos al plano XZ.

Es un método sencillo ya que solo se sustituirán las variables Y de los puntos por 0 mediante la utilización de las funciones públicas contenidas en la clase ProjectInliers. Viendo la Figura 40 desde el plano XZ tenemos:

Figura 41: Plano XZ previa proyección.

En la Figura 41 vemos que todos los puntos no forman parte de un plano perfecto. Aplicamos la proyección y tenemos el resultado expuesto en la Figura 42:

Figura 42: Plano XZ una vez proyectado.

Sin haber sufrido ningún cambio en cuanto a forma del sector transitable.

5.8. Obtención de obstáculos elevados sobre el suelo

Hasta este punto hemos tenido en cuenta los obstáculos que tuviesen base y que esa base o parte baja del obstáculo fuese la más grande es decir, que ese elemento tuviese forma cilíndrica como máximo.

Se debe prever la posibilidad de encontrar obstáculos como la barrera de la Figura 43 que no tuviesen una base definida sobre el suelo o de elementos que tuviesen aspecto cónico siendo la parte baja más pequeña que la parte alta.

Figura 43: Ejemplo de obstáculo conflictivo.

Para evitar el conflicto con dichos elementos utilizaremos la información que eliminamos al segmentar los puntos que no formaban parte del plano (Sección 5.5). Con ello tendremos todos los posibles obstáculos, los que no pudimos apreciar al segmentar el plano y los que ya estaban eliminados.

Utilizaremos el mismo algoritmo que en la extracción del plano (Sección 5.5) pero esta vez, a la hora de guardar los inliers, lo que se hará será eliminar estos y quedarnos con la parte restante. Con ello tendremos únicamente los obstáculos que es la parte buscada y que posteriormente restaremos para tener un plano resultante total (Sección 5.10).

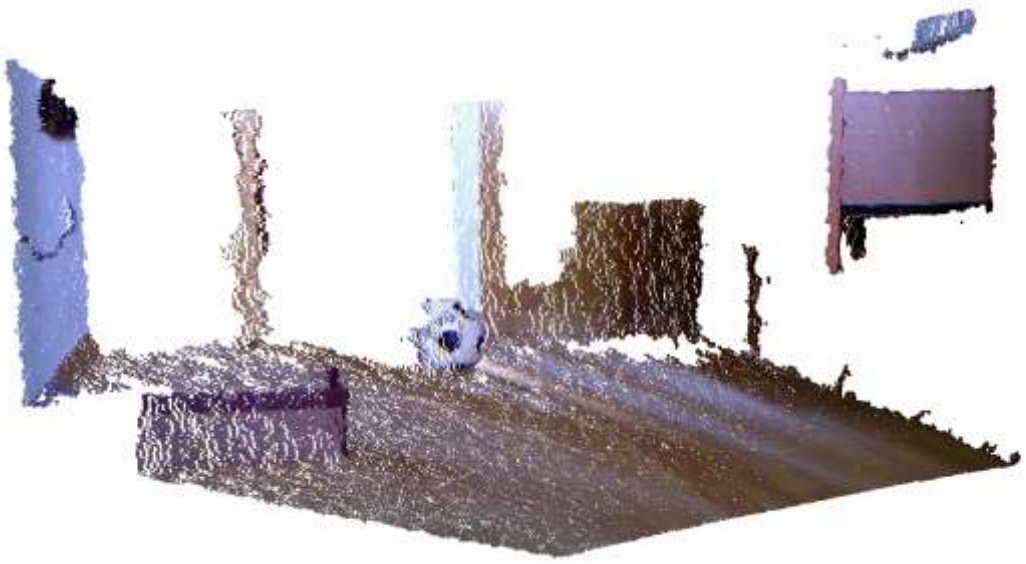


Figura 44: Imagen inicial de la estancia.

Dada la imagen de la Figura 44 donde vemos de nuevo la estancia inicial, segmentamos los objetos superiores.

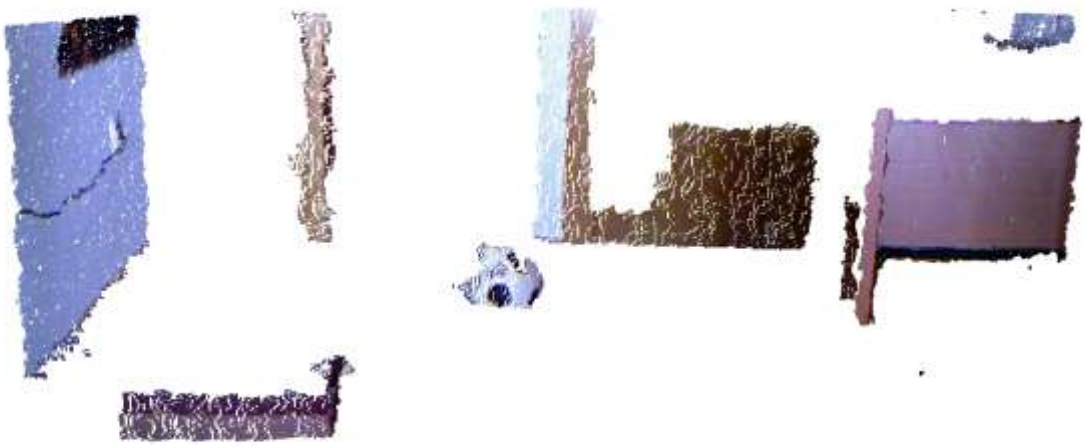


Figura 45: Nube de puntos de los obstáculos elevados sobre el suelo.

Obtenemos únicamente los obstáculos de la Figura 45, es decir la parte que no entra dentro del plano obtenido dentro de los inliers.

5.9. Proyección de obstáculos elevados

Una vez tenemos los obstáculos los proyectamos sobre el mismo plano que el suelo anteriormente (Sección 5.7). Con esto conseguiremos tener todo sobre la misma superficie de trabajo.

A partir de la Figura 45 realizaremos la proyección teniendo como plano resultante: resultante el mostrado en las imágenes 46 y 47:

Figura 46: Proyección de los obstáculos elevados.

Figura 47: Vista próxima al plano XZ de la proyección.

El resultado como vemos son los obstáculos obtenidos en la Sección 5.8 pero trasladados al plano de trabajo utilizado también en la Sección 5.7.

5.10. Resta de planos y guardado del plano resultante

Una vez tengamos los dos conjuntos de puntos necesarios, restaremos el formado por los obstáculos al suelo teniendo así un plano final totalmente libre que se corresponderá al plano transitable que buscábamos.

Para evitar posibles errores en la resta llevaremos a cabo con un margen de seguridad de aproximadamente 3cm. Gracias a ello, a la hora de restar, se crea un margen vacío alrededor de los puntos eliminados que nos permitirá tener un margen de seguridad.

Para llevar a cabo esta segmentación utilizaremos un objeto de la clase SegmentDifferences (Sección 4.4) donde estableceremos una imagen base sobre la que restar (Figura 48) y su resta (Figura 49) realizando la operación Base-Resta. Tendremos de esta manera el resultado (Figura 50) que mostrará la superficie del suelo de manera clara y libre de cualquier obstáculo que pueda obstruir el paso de nuestro robot por la estancia tratada.

Dado un plano Base como el que podemos ver en la Figura 48 donde podemos ver la superficie del suelo si los obstáculos apoyados sobre esta superficie:

Figura 48: Plano sin obstáculos que tienen la base marcada.

Si segmentamos los obstáculos siguientes:

Figura 49: Plano de obstáculos.

Tendremos como resultado el plano definitivo en la Figura 50:

Figura 50: Plano definitivo.

Podemos apreciar como el vacío que aparece en la parte inferior derecha de la imagen contiene el segmento no transitable generado por el cajón con un margen mayor al que de verdad ocuparía la parte del cajón

6. Resultados Experimentales

En este capítulo se presentarán diferentes pruebas realizadas con el algoritmo anteriormente descrito en distinta situaciones. Probaremos el software para obtener sus distintas posibilidades y limitaciones. Mediremos el tiempo empleado en cada una de las pruebas para llevar mostrar el coste computacional del algoritmo.

6.1. Visionando paredes

Ante la posibilidad de que el robot portador de la cámara Kinect se encuentre frente a una pared a una distancia cercana tal que las limitaciones de rango de visión no le permitan captar superficie del suelo (Figura 51):

Figura 51: Imagen de visión de pared.

El programa nos permitirá darnos cuenta de la situación ya que la salida de la segmentación de la zona perpendicular al plano Y nos dará 0 puntos al no poder devolver suficientes inliers que cumplan esa condición (Figura 52).

```
Imagen cargada correctamente con: 307200 puntos.
Tiempo 2071.26 ms
Imagen recortada en Y (suelo) con un resultado de : 255736 puntos. YSUELO
Tiempo 846.689 ms
Imagen recortada en Z (suelo) con un resultado de : 255736 puntos. ZSUELO
Tiempo 831.723 ms
[pcl::SampleConsensusModelPlane::optimizeModelCoefficients] Not enough inliers f
ound to support a model (0)! Returning the same coefficients.
El plano perpendicular al eje Y con una variacion de 0.2 gradianes tiene: 0punto
s. GRADSUELO
```

Figura 52: Mensaje de error al no encontrar suelo.

Como podemos ver, la imagen se recorta en Z e Y pero no puede realizar mas acciones al no disponer de un plano perpendicular al eje Y con el que poder trabajar.

6.2. Visionando paredes y suelo

Para comprobar el buen funcionamiento del algoritmo en caso de percibir un plano de pequeño tamaño (suelo) y otro superior (pared) tenemos una imagen como la mostrada en la Figura 53:

Figura 53: Imagen de visión de pared con suelo.

Se ha realizado esta prueba ya que, al tener un plano superior, el algoritmo podría fijar como plano correcto el que tuviese mayor información. La introducción del algoritmo de extracción de la zona perpendicular al eje Y nos permite evitar posibles fallos por lo que el resultado es el correcto al obtener la siguiente nube de puntos:

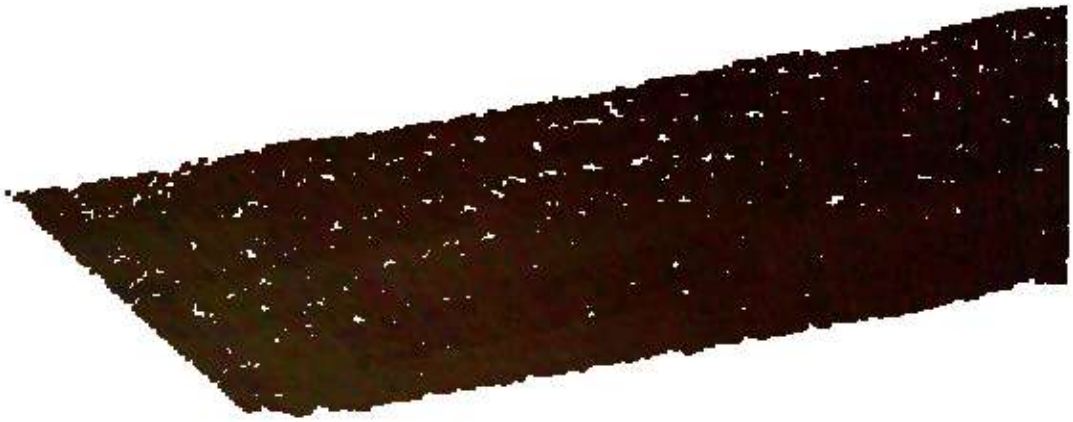


Figura 54: Extracción correcta del suelo.

El programa secciona correctamente la imagen, y extrae el plano del suelo correctamente. Al no detectar ningún objeto el resultado final es el obtenido en la Figura 54.

El tiempo total para la realización de esta prueba es de: 7.321 segundos.

6.3. Visionando sin luminosidad

Una ventaja significativa que dispone esta cámara es el haz infrarrojo que le permite tomar imágenes con profundidad a pesar de no tener visibilidad. Realizando una prueba para ver el alcance de esta capacidad tenemos la siguiente imagen mostrada en la Figura 55 en la que vemos una estancia con un objeto central que impediría el paso, por lo que debería ser tratado como un obstáculo.

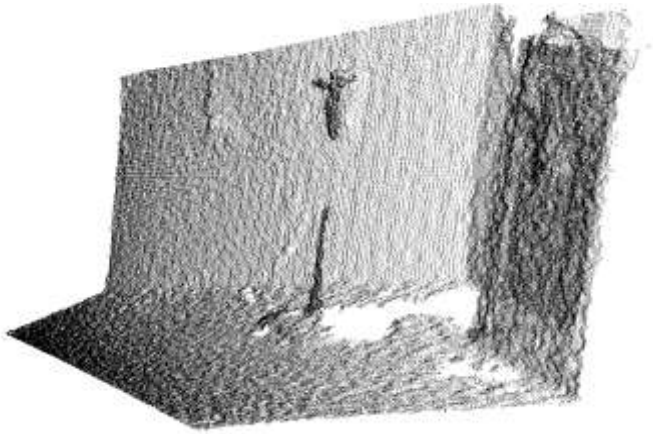


Figura 55: Estancia sin luminosidad.

Llevando a cabo el programa sin realizar ninguna modificación especial, pese a las condiciones de escasa visibilidad en las que se trabaja, tenemos el plano del suelo con el sector correspondiente al obstáculo extraído (Figura 56).

Figura 56: Resultado de la superficie sin obstáculos.

El programa ha realizado correctamente la prueba llevando a cabo cada uno de los pasos vistos en el Capítulo 5.

El tiempo total para la realización de esta prueba es de: 9.177 segundos.

6.4. Visionando con múltiples obstáculos

Para realizar la siguiente prueba presentamos una estancia con múltiples obstáculos (Figura 57). Se ha de remarcar la presencia de dos de ellos más adelantados (cajón y soporte en la parte delantera de la imagen) que distan 40 centímetros entre sí.

Figura 57: Estancia con múltiples obstáculos.

Llevando a cabo una correcta segmentación veremos como se han extraído correctamente los objetos que podrían obstaculizar el paso. Se ha de remarcar como se percibe en la Figura 58 la aparición de grandes vacíos correspondientes a objetos que, a priori, son más pequeños.

Figura 58: Resultado final del plano cruzable.

Gracias a este margen tendremos un resultado fiable reduciendo las posibilidades de colisión.

El tiempo total para la realización de esta prueba es de: 11.015 segundos.

6.5. Visionando con habitación libre

En este ejemplo pondremos a prueba la fiabilidad del programa en un entorno menos complicado donde tengamos una habitación libre (Figura 59).

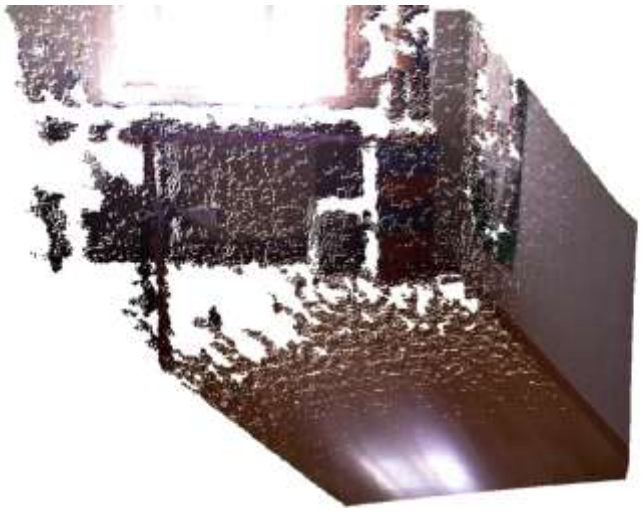


Figura 59: Estancia libre de obstáculos.

La prueba se realiza de manera correcta sin ningún fallo a través de los distintos pasos que desembocan en un resultado libre de obstáculos (Figura 60).

Figura 60: Plano resultante libre de obstáculos.

El plano corresponde al suelo de la habitación hasta una distancia de 2.65 metros aproximadamente (limite fijado en el la Sección 5.3).

El tiempo total para la realización de esta prueba es de: 7.892 segundos.

6.6. Visionando en estancia muy amplia con objetos

En la siguiente prueba tomaremos como entorno una estancia muy amplia con diversos objetos como la de la Figura 61.

Figura 61: Entorno amplio y con objetos.

El resultado tras la segmentación debería de ser la parte delantera de la imagen de la cual se segmentará posteriormente los objetos.

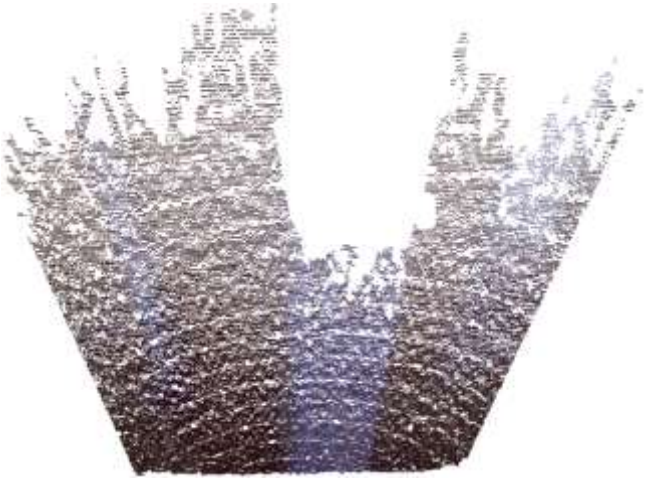


Figura 62: Resultado de la segmentación.

El resultado esta reflejado en la Figura 62, donde vemos el suelo de la estancia libre de manera correcta.

El tiempo total para la realización de esta prueba es de: 10.113 segundos.

6.7. Visionando en estancia con reflejos

En la siguiente prueba tomaremos como entorno una estancia con gran visibilidad y un suelo mas pulido que refleje la luz con mayor intensidad (Figura 63).

Figura 63: Estancia amplia con suelos que reflejan la luz.

El resultado obtenido es el correcto como podemos observar en la Figura 64. La cámara no ha sufrido ninguna variación al estar ante una superficie más brillante.

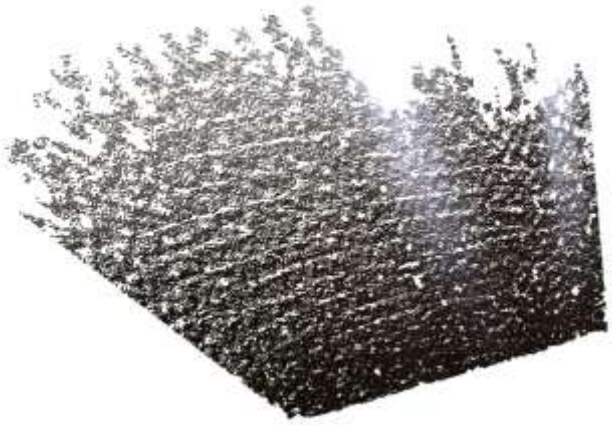


Figura 64: Resultado del suelo con gran reflejo.

El tiempo total para la realización de esta prueba es de: 9.737 segundos.

6.8. Visionando objetos con posibles errores

En la siguiente prueba tomaremos como entorno una estancia con objetos que, al estar elevados, permiten que podamos visualizar superficie debajo de ellos (Figura 65).

Figura 65: Imagen con objetos elevados.

Esto puede llevar a confusión ya que nuestro programa podría tomar como transitable la parte superficial correspondiente al segmento de suelo que esta por debajo de los objetos.

Figura 66: Resultado del mapeado conflictivo con objetos elevados.

En la Figura 66 podemos ver la buena respuesta de este programa ya que ha reconocido el suelo que estaba situado debajo de los objetos, pero ha establecido las zonas no transitables adelantadas a este espacio. Esto permite determinar que esas zonas no son cruzables ya que no se puede acceder a ellas sin pasar "atravesando" el obstáculo.

El tiempo total para la realización de esta prueba es de: 8.468 segundos.

7. Conclusiones

En este apartado analizaremos el trabajo realizado, las dificultades encontradas, los aspectos positivos, etc.

El objetivo principal del proyecto, la estimación de zonas cruzables utilizando la cámara Kinect desarrollando una aplicación en C++ se ha cumplido de manera correcta.

Se ha obtenido un programa que, tras una captura de la información proveniente de un entorno cualquiera, se puedan obtener las zonas cruzables evitando todos los obstáculos existentes. Esto reduce considerablemente el riesgo de colisiones del sistema portador de este programa.

La principal ventaja que tenemos es la fiabilidad de resultados de la que dispone ya que segmenta objetos con un margen de error dando seguridad ante el resultado obtenido. Además tiene la posibilidad de obtener resultados con visibilidad nula.

Entre sus desventajas encontramos el gran tiempo necesario para procesar nubes de puntos con mucha información. Esto lastrará la fiabilidad de resultados en un mundo dinámico como en el que vivimos. El rango de visión del dispositivo empleado restringe mucho los resultados ya que limita la capacidad de ver principalmente objetos muy próximos.

Gracias a este proyecto he podido tomar contacto con la visión artificial de manera directa. He podido darme cuenta de hasta que punto, sensores al alcance de cualquiera, pueden tener tanto potencial en la automatización y la robótica.

He podido enriquecer mis conocimientos con modelos de búsqueda de propiedades como el método KdTree, además de algoritmos de reconocimiento de formas o modelos como es el método RANSAC.

Entre las dificultades que he encontrado, he de remarcar mi desconocimiento hacia una librería con tanto potencial como es PCL. He aprendido ha desenvolverme en un SO como es Ubuntu donde no había trabajado antes y que durante los primeros meses me fue bastante lioso.

Remarcaré también la gran aportación de este proyecto en el ámbito más personal. Gracias a este trabajo he aprendido como limitaciones que veía claras para mí, como era la programación, he podido superarlas a base de constancia y trabajo. El trabajo día a día, compaginándolo con el desarrollo de un curso tan complicado y duro en cuanto a carga de trabajo se refiere, me ha ayudado a quitarme el miedo a enfrentarme a cualquier problema.

He de agradecer la labor de mi tutor, por guiarme a través de este campo que anteriormente no conocía demasiado, pero también por darme libertad creativa además de mucho margen para plantear la resolución de los problemas de manera personal.

Por último agradecer también todo el tiempo a mi familia y amigos que me han dado fuerzas en todo momento.

7.1. Mejoras y líneas de trabajo futuro

El programa funciona de manera eficiente dando buenos resultados a múltiples experimentos que se han ido realizando, muchos de los cuales están incluidos en el Capítulo 6.

Dentro de las posibles mejoras del programa podríamos encontrar:

- Mejora de la eficiencia de los algoritmos en cuanto a tiempo de ejecución.

Algunos de los algoritmos emplean tiempo excesivo (Figura 67) a la hora de realizar distintos pasos como el de segmentar secciones de la nube de puntos.

```
Imagen cargada correctamente con: 307200 puntos.
Tiempo 2000.54 ms
Imagen recortada en Y (suelo) con un resultado de : 190901 puntos. YSUELO
Tiempo 812.641 ms
Imagen recortada en Z (suelo) con un resultado de : 182838 puntos. ZSUELO
Tiempo 753.606 ms
El plano perpendicular al eje Y con una variacion de 0.2 gradianes tiene: 80281puntos. GRADSUELO
Tiempo 2131.58 ms
El plano segmentado tiene: 67209puntos. PLANOSUELO
Tiempo 1209.02 ms
Suelo proyectado: OK PROYSUELO
Tiempo 297.328 ms
El plano perpendicular al eje Y con una variacion de 0.2 gradianes tiene: 102557puntos. GRADBARRERA
Tiempo 2120.12 ms
Barrera proyectada: OK PROYBARRERA
Tiempo 396.578 ms
El plano sin obstaculos tiene tiene: 54421puntos. TOTAL
Tiempo 560.047 ms
Tiempo total ejecucion: 10281.8 ms
```

Figura 67: Tiempos de ejecución del programa con una imagen genérica.

Esto implica una gran dificultad de emplear dicho trabajo para acciones en tiempo real.

- Mejorar las capacidades de nuestra cámara Kinect.

Nuestro dispositivo a pesar de ser un buen sensor, barato y versátil, debería de obtener mejores resultados a larga distancia reduciendo su error. Además seria muy importante reducir la limitación de visión, sobre todo en distancias cortas ya que nos permitiría una previsión de colisiones de corto alcance mucho mejores.

Dentro de las líneas de trabajo futuro, podríamos centrarnos en poder trabajar con las nubes de puntos resultantes dentro de mallados de ocupación para poder llevar a cabo, con programas como Matlab, un algoritmo de estimación de trayectorias. Además se podría estudiar la incorporación de este programa a otros como el de reconocimiento de objetos para dotar a un sistema robótico de la capacidad de moverse e interaccionar con diferentes objetos de su entorno. Dichos robots podrían servir como robot de servicios para personas con discapacidades visuales, ya que podrían "ser sus ojos".

Bibliografía

1. Hebert, M., Thorpe, C. E., & Stentz, A. (1997). Intelligent unmanned ground vehicles: Autonomous navigation research at Carnegie Mellon. Boston: Kluwer Academic Publishers.

2. Ünsal, C. (1997). Intelligent navigation of autonomous vehicles in an automated highway system: Learning methods and interacting vehicles approach. Blacksburg, Va: University Libraries, Virginia Polytechnic Institute and State University.

3. Vandapel, N., Donamukkala, R., & Hebert, M. (January 01, 2006). Unmanned Ground Vehicle Navigation Using Aerial Ladar Data. The International Journal of Robotics Research, 25, 1, 31-51.

4. LASSERRE, Patricia. & BRIOT, M. (1996). VISION POUR LA ROBOTIQUE MOBILE EN ENVIRONNEMENT NATUREL.

5. Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, Mahoney, P. (January 01, 2006). Stanley: The robot that won the DARPA Grand Challenge. Journal of Field Robotics, 23, 9, 661.

6. Kaufman, M. (2012). Mars landing 2012: Inside the NASA Curiosity Mission. Washington, D.C: National Geographic Society.

7. Shoop, B., Johnston, M., Goehring, R., Moneyhun, J., Skibba, B., & Product Manager Force Protection Systems. (2006). *Mobile Detection Assessment and Response Systems (MDARS): A force protection, physical security operational success*. Fort Belvoir, VA: Defense Technical Information Center.

8. Laboratorio de robótica Universidad Carlos III Madrid. Imágenes e información. Disponible en Web: http://roboticslab.uc3m.es/roboticslab/robot.php?id_robot=5

9. Labayrade, R., & Aubert, D. (January 01, 2004). Robust and fast stereovision based obstacles detection for driving safety assistance. Ieice Transactions on Information and Systems, 1, 80-88.

10. OpenNI. Documentación e información. Disponible en Web: http://openni.org/documentation/

11. Biblioteca PCL. Documentación y tutoriales de ayuda. Disponible en Web: http://pointclouds.org/documentation/

12. MALIK, D. S. (2012). C++ programming: Program design including data structures. s.l.: Wadsworth publishing

13. Alfonseca, M., & Alcalá, A. (1992). Programación orientada a objetos: teoría y técnicas OPP para desarrollo de software. Madrid: Anaya Multimedia

14. Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1975). An algorithm for finding best matches in logarithmic time. Stanford, Calif: Dept. of Computer Science, Stanford University.

15. Bentley, J. L., & Carnegie-Mellon University. (1978). Multidimensional binary search trees in database applications. Pittsburgh, Pa: Departments of Computer Science and Mathematics, Carnegie-Mellon University.

16. Fischler, M. A., Bolles, R. C., & Stanford Research Institute. (1980). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Stanford, Calif.: SRI International.

17. Leavers, V. F. (1992). Shape detection in computer vision using the Hough transform. London: Springer-Verlag.